University of Canterbury

COSC460 Honours Report

Algorithms for Real-Time Rendering of Soft Shadows

Richard Viney

rjv20@student.canterbury.ac.nz

8th November, 2007

Supervisor: Dr R. Mukundan

mukundan@canterbury.ac.nz

Abstract

In computer graphics, calculating realistic shadowing and lighting terms for an arbitrary scene is a fundamental problem. When accurate interaction of lights with objects in a scene is achieved it greatly enhances the believability and the immersion that the viewer experiences. In this report we give a background to real-time shadow determination algorithms and present two approaches for real-time rendering of shadows that accurately model the umbrae and penumbrae of area light sources. A description of geometry shaders, a new technology in real-time rendering, is given, and we describe new methods that demonstrate how they can be used to shift significant amounts of mesh computation from the CPU to the GPU in the penumbra wedge soft shadow algorithm. We also present comparative and performance analyses of the soft shadow algorithms developed and discuss the performance characteristics of utilizing the geometry shader stage in shadowing algorithms. Our geometry shader based implementation provides a 21% performance increase to the penumbra wedge soft shadowing algorithm for certain meshes.

Keywords: soft shadows, geometry shader, penumbra wedge, real-time

Acknowledgements

Thank you to my research supervisor, Dr. Ramakrishnan Mukundan, for supervising this Honours project and giving me the benefit of his experience and guidance throughout the year.

Thank you to Savant Information Systems for providing the Carbon Game Engine as a foundation for conducting this research.

Thank you also to all the people who donated their time to proofread and fact-check this report, without their input it would likely not have reached its full potential.

Finally, thank you to all those people who have supported me throughout the 2007 academic year, especially to my parents for their constant support and encouragement.

Contents

List of Figures

List of Code Listings

List of Tables

Chapter 1

Introduction

In computer graphics, calculating realistic shadowing and lighting terms for an arbitrary scene is a fundamental problem. When accurate interaction of lights with objects in a scene is achieved it greatly enhances the believability and the immersion that the viewer experiences. It is therefore important that the methods and algorithms used to simulate these phenomena are being continually improved and refined to improve their accuracy, speed, and subjective visual quality.

In this introduction we cover a number of basic concepts of relating to shadows, including why they are important, how they are formed, and how humans recognize and process shadows in visual imagery. All these factors play a key part in our motivation for modeling this physical phenomenon in real-time rendering. The introduction concludes with a structural outline of the remainder of the report.

1.1 Role of Shadows

Shadows provide vital visual clues that communicate spatial relationships and information to the viewer. Often these relationships are not communicable through any other visual means which makes it particularly important that they are represented accurately.

Figure 1.1 illustrates one of the types of information that is often contained in shadows. In this instance the shadow communicates to the viewer that the robot is getting further away from the floor, without the shadow this would not be apparent. Shadows can also communicate properties of objects that are hidden from direct view, such as when the shadow cast by an object contains details of the object that would otherwise be entirely unknown to the viewer. Consider the case where there is a set of objects arranged next to each other in a line, with a viewer at one end of the line. Assuming the objects are all the same shape the viewer would only be able to see the object closest to them, and would have no way of knowing how many objects were in the line, or even that there were multiple objects present. However, a light positioned in front of the line of objects could cast shadows that the viewer can then use to deduce the number of objects that are present and their approximate positions, despite the fact that none of the objects are directly visible.

Figure 1.1: Shadows contain much important information about a scene, such as the distance of an object from the floor. Source: [1].

1.2 Shadow Optics

Before looking at the algorithms used to generate shadows in a digital environment it is important to understand the physical basis for the phenomena being modeled. In the real world, the edges of shadows cast by objects are typically characterized by smooth light to dark transitions. These transitions are the result of a light source being partially occluded. This is illustrated in Figure 1.2.

The dark central area of a shadow is called the umbra, and is the area where no light from the light source is able to reach along a direct path from the light source. The outer area where the smooth transition from light to dark occurs is called the penumbra. Moving away from the umbra through the penumbra results in progressively more and more of the light source becoming visible, and this gradual increase in illumination is what gives shadows their soft edges. Note that if the light source were to get larger then the penumbra area would also enlarge, resulting in a shadow with edges that are smoother and have reduced definition. If the light source becomes large enough it is possible for the umbra to disappear from the receiving surface entirely, resulting in an extremely fuzzy and undefined shadow.

Accurately modeling the visual properties of umbrae and penumbrae is the central challenge for realistic shadowing algorithms.

Light sources in computer graphics have traditionally been modeled as single points in space with no assumed volume, these are referred to as *point light sources*. This results in the generation of so-called *hard shadows* in which the penumbra has zero volume. This can be verified by extrapolating Figure 1.2 to the point that the radius of the light source is zero. Hard shadows are characterized by an instantaneous transition from shadowed to non-shadowed. That is, each pixel in the final image has a binary classification of being either in or out of shadow, with no accommodation for

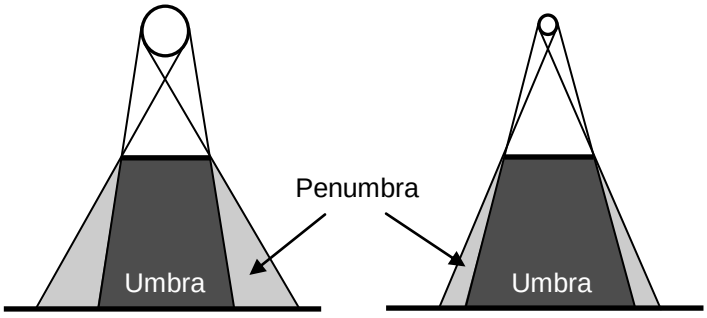

Figure 1.2: The umbra and penumbra of a simple two dimensional shadow. Two different light source sizes are given for comparison. Note that the size of the penumbra is a function of the size of the light source, the distance from the light to the shadow casting object, and the distance from the shadow casting object to the shadow receiver.

intermediate transition states that would allow penumbrae to be modeled.

The primary reason for the popularity of hard shadow algorithms has been their real-time performance characteristics that have allowed them to be used on a wide variety of scenes. However, as graphics hardware has become significantly more computationally capable over the last 5 years the possibility of moving to more realistic calculation of penumbrae detail that results in more visually pleasing and physically accurate shadows has received a lot of attention from researchers, with a variety of different algorithms being proposed [2, 3, 4, 5, 6, 7, 8].

The term *soft shadows* is used to describe the type of shadow generated by these algorithms. They all provide a mechanism for modeling *area light sources*, which are light sources that have volume in the scene and therefore entail the modeling of penumbrae. The presence of penumbrae means that the resultant shadows will have soft edges, hence the term 'soft shadows'. These algorithms are the focus of this report and are covered in detail in Chapter 3.

1.3 Human Shadow Perception

Organizing and processing all the raw visual stimuli generated by the eyes requires a massive amount of cognition, and humans typically draw on a huge library of prior knowledge and experience when attempting to make sense of a new stimulus. Separation of shadows from objects is a largely unconscious mental process and occurs very rapidly.

One property of human shadow perception that is particularly relevant is how undemanding we tend to be when evaluating a shadow. The inherent imprecision and fuzziness of most shadows means that the fine details of the original object are typically not present in its shadow, however this does not usually adversely affect recognition accuracy provided the overall shape is left intact. Research on soft shadows in virtual environments has given the surprising result that "a mesh simplified to only 1% of its original complexity is capable to cast soft shadows that satisfy 90% of test persons"

Figure 1.3: Shadow error caused by reducing mesh complexity. The shadow in the left image is cast by the full resolution mesh, and the shadow in the right image is cast by a mesh with 10% of the triangle complexity. Interestingly, the shadow in the right image was acceptable to 90% of people surveyed [9].

[9]. Earlier research also found that people are generally poor judges of the correctness of shadow shape [10], and drew the conclusion that generated shadows need only be plausible instead of physically correct. Figure 1.3 illustrates this result.

Many inferences about scenes and objects are made from observing shadows. It has been shown that adjusting the appearance of the shadow cast by a moving object can result in a dramatic shift in the object's perceived trajectory [11, 12].

Given that shadows are extremely important for interpreting images, it is not surprising that human perception of shadows also presents some fascinating results in the form of shadow-based optical illusions. There are many well-known optical illusions of this type, two of which are demonstrated in Figure 1.4. The first image is the famous "checkerboard shadow illusion" which demonstrates that human color perception is strongly influenced by the color of surrounding objects and lighting conditions, the two squares labeled *A* and *B* are in fact the same color. The second image is an example of the brain extrapolating backwards from a shadow to an object, which in this case is two people sitting back to back, and coming to entirely the wrong conclusion: the shadow is in fact cast by a well-crafted pile of debris.

A lot of psychological theory, such as the well-known set of Gestalt laws, has been created to explain and characterize these and other similar types of phenomena relating to human visual perception. These phenomena give a fascinating and relevant insight into how the shadows produced by the algorithms discussed in this paper will subsequently be interpreted by the viewer, and this information can be applied to achieve effective shadow quality scaling by reducing the shadow quality in the way that has the least affect on subjective visual quality as seen by human eyes [9].

It is also important to realize that there is no 'perfect' shadow. When comparing shadowing algorithms quantifiable measurements such as performance and scalability are important, but overall visual quality is a fundamentally subjective measure based on inherently fuzzy human visual perception. It is quite possible that technically incorrect shadows, that is, ones that are at variance with what would be seen in reality, may in fact be acceptable to the majority of viewers [9, 10].

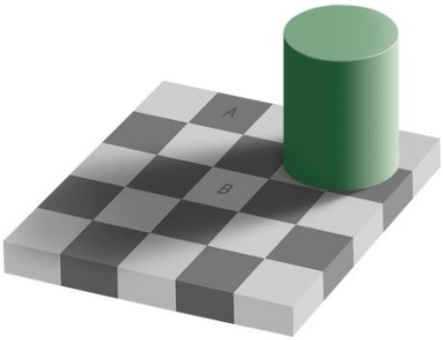

Figure 1.4: Shadow-based optical illusions. The left image is the famous checkerboard illusion, squares A and B are in fact the same color. The right image is a design by Shigeo Fukada that shows how shadows can provide misleading information about objects.

1.4 Report Outline

The remainder of this report is structured as follows. In Chapter 2 we review existing algorithms for real-time shadow generation; in Chapter 3 we describe several key algorithms for the generation of soft shadows in real-time; in Chapter 4 we present novel improvements to these algorithms using features present in the latest generation of graphics hardware; Chapter 5 contains the results of a comparative analysis of the soft shadowing algorithms in this report, including our improvements from Chapter 4; finally, Chapters 6 and 7 look at potential avenues for further work, and conclude our research.

Chapter 2

Background & Related Work

Several comprehensive publications are available that cover the progression of shadow generation algorithms over the last 20 years. The papers by Woo *et al.* [13] and Hasenfratz *et al.* [1] are the best of these, and a comparison between these two publications shows the tremendous progress made in the intervening decade. This progress was made possible largely by the rapid improvement and uptake of consumer-level graphics processors. Despite real-time 3D graphics being commonplace for well over a decade, incorporating shadows into real-time applications remains a significant challenge that has generated a huge research effort.

This chapter reviews the key algorithms for calculating shadows in real-time applications, the principles of which form the core of new algorithms for generating soft shadows that are discussed in Chapter 3.

2.1 Requirements

We are concerned with the generation of dynamic soft shadows in real-time. Off-line pre-computation of shadowing terms using techniques such as ray-tracing, radiosity [14] and photon mapping [15] are therefore not considered, although they should be evaluated in cases where dynamic shadowing is not required as they can generate exceptionally realistic real-time images. We define a rendering algorithm as real-time if it can achieve an update rate of at least 30Hz at an output resolution of 640x480 on currently available hardware when used with a single light source and a scene of at least 5000 triangles. At the time of writing the highest performing graphics processor available was the NVIDIA GeForce 8800 Ultra.

In addition to real-time performance we also require that algorithms permit every object in the scene to be dynamic, including light sources. All objects, including those that are receiving and/or casting shadows, must be allowed to undergo any arbitrary rotational or translational transformation whilst maintaining real-time performance. Additionally, algorithms should not restrict what geometry configurations can receive or cast shadows, ruling out algorithms such as projected planar shadows [16] and Haines' plateau algorithm [17] as they can only cast shadows onto flat planar surfaces. We do however permit algorithms to stipulate that input geometry conforms to certain requirements, such as that it is 2-manifold or satisfies some other well-defined geometric

criteria.

The final requirement is that all geometric objects be allowed to alter their shape through the use of kinematic systems such as skeletal animation or vertex tweening, and that calculated shadows will be altered appropriately without losing real-time performance. Note that we are only concerned that these requirements be met by the shadowing algorithm when considered in isolation, obviously having a significant amount of dynamic geometry will place other computational burdens on the system outside of shadowing calculations, but we do not consider these costs relevant to shadowing algorithm performance. Despite this requirement, it is considered an excellent advantage for an algorithm to have the ability to significantly improve its performance when used with geometry or light sources that are defined as being static in the scene.

This set of requirements significantly reduces the number of potential algorithms for calculating real-time shadows, and in this chapter we review the most influential of such algorithms.

2.2 Shadow Mapping

At its most basic level, computing shadows is the task of identifying the elements of the scene that are hidden from the light source, which can also be solved by finding the scene elements that are visible from the light source's point of view. The shadow mapping algorithm determines visibility using exclusively image space techniques, which means that it does not require any knowledge of the structure of scene geometry. The only requirement is that shadow casting geometry is able to be rendered.

Shadow mapping is a two pass algorithm. The first pass is the rendering of a shadow map that will be used to determine whether a given point in the scene is in or out of shadow. The shadow map is a two dimensional buffer created by rendering the scene from the light source's point of view. Each pixel in the shadow map holds the distance to the nearest object seen through that pixel as observed from the light source, which is the same as the final state of the z-buffer following rasterization of all visible triangles. This relationship with z-buffering is important because z-buffering is widely used and is supported in hardware by every consumer graphics processor made in the last decade, and can therefore accelerate the creation of the shadow map. The top row of images in Figure 2.1 illustrates the creation and content of a shadow map in a simple scene.

Shadow determination is done by rendering the scene from the viewer's point of view. For each rasterized fragment a binary shadow term is found by determining the fragment's (x,y,z) position in light space, looking up the shadow map pixel at position (x,y), and then comparing this value to the fragment's light space z position. This comparison therefore has the following two inputs:

1. A, the z value read from the shadow map at the fragment's light space (x,y) position
2. B, the z value of the fragment's light space position

If B is greater than A then there must be some object closer to the light than the fragment, so therefore the fragment is in shadow. Similarly, if A and B are approximately equal then the fragment is illuminated by the light. In a correct shadow mapping implementation it is not possible for A to be greater than B, although such a result still usually

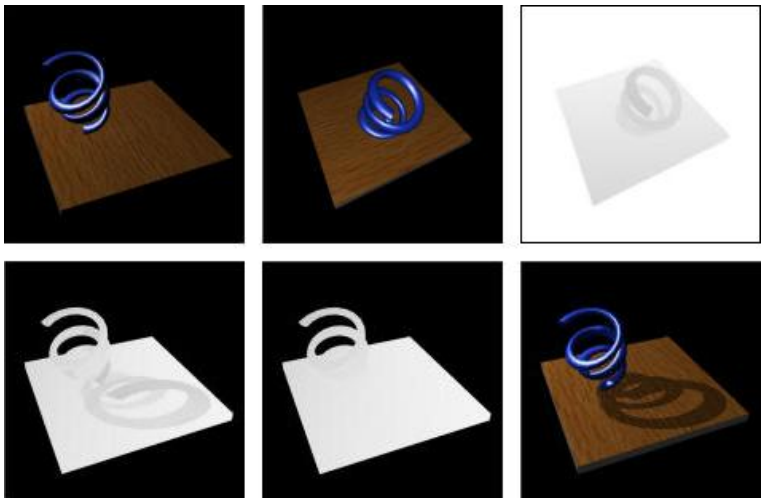

Figure 2.1: The shadow mapping algorithm. From top left: the unshadowed scene from the viewer's point of view; the unshadowed scene from the light source's point of view; the shadow map from the light's point of view; the shadow map projected onto the scene, seen from the viewer's point of view (the A value); the z distance to the light source (the B value); the final scene from the viewer's point of view.

results in illumination of the fragment. This shadow determination step is described by the pseudo-code in Listing 2.1.

Listing 2.1: Pseudo-code for the shadow determination stage of the shadow mapping algorithm.

```
for all rasterized fragments
    P = fragment xyz position in light space
    A = shadow map value at (P.x, P.y)
    B = P.z
    if A < B then
        fragment is in shadow
    else
        fragment is illuminated
    end if
end for
```

The result of the shadowing determination is used to mask out lighting calculations, resulting in the appearance of shadow on the surface. All of these per-fragment operations, including the depth comparison, are fully hardware accelerated on most graphics processors, so the speed of shadow mapping is usually suitable for real-time applications.

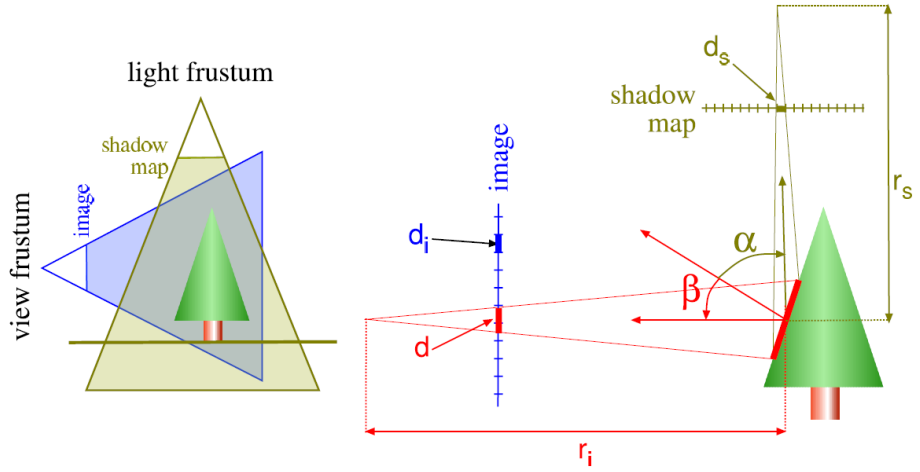

Figure 2.2: Shadow map projection quantities.

2.2.1 Evaluation

The shadow mapping algorithm has a number of drawbacks. The discrete sampling of scene depths present in the shadow map results in aliasing artifacts, the severity of which depends primarily on the resolution of the shadow map. Also, the projection of the shadow map onto the scene can result in a single shadow map pixel being used for shadow determination on a large number of pixels in the final image which results in unpleasing shadows with a blocky appearance around the edges. Higher resolution shadow maps will result in each shadow map pixel projecting to a smaller volume of the final scene, resulting in more accurate shadow determination and fewer of these types of aliasing artifacts.

Aliasing in shadow mapping is caused by undersampling, that is, the set of discrete samples present in the shadow map becomes too sparse to allow accurate shadow determination. With reference to Figure 2.2, this is described mathematically by the following equation:

$$d = d_s \frac{r_s}{r_i} \frac{\cos\beta}{\cos\alpha} \tag{2.1}$$

This equation gives the approximate pixel area d that a shadow map pixel of size d_s will correspond to given the distance from the light to the surface, r_s, the distance from the eye to the surface, r_i, the angle from the surface normal to the light source, α, and the angle from the surface normal to the eye, β. Undersampling occurs when d is larger than the size of the image pixels, d_i, although typically d/d_i needs to be at least 2 for the resultant aliasing to be noticeable. Undersampling can happen for two separate reasons. The first is when $d_s r_s / r_i$ becomes large, which typically occurs when the viewer moves very close to the surface receiving shadow, that is, $r_i \to 0$. The second cause of undersampling is when $\cos\beta / \cos\alpha$ becomes large, which occurs when the shadow receiving surface is nearly in line with the direction of projection of the shadow map pixel, that is, $\cos\alpha \to 0$.

Figure 2.3: The effect of percentage closer filtering on shadow map quality. The shadow in the right image has been filtered while the shadow in the left image has not.

When transforming the position of an illuminated fragment from world space to light space the resultant z value would ideally exactly match the z value read from the shadow map, however due to quantization of z values this is rarely the case, and the transformed z value may fall slightly above or below the surface depth read from the shadow map. If the transformed z value falls below the surface it causes erroneous self-occlusion on the illuminated surface. The solution to this problem is to use a bias value that has the effect of slightly insetting the depth of the shadow casting surface so that these quantization errors are eliminated when performing depth comparisons.

The final major drawback of shadow maps is their inability to handle omnidirectional point light sources, however they are a natural fit for other type of lights such as spotlights, where light is cast out in a restricted area. Omnidirectional light sources are problematic because there is no way to render every object that is visible in any direction from a single point, meaning a shadow map can't be created. To use shadow mapping with this kind of light source requires it be split up into a number of component light sources each covering a subset of the total field-of-view of the omnidirectional light source. Work has been done on reducing the number of component light sources required with techniques such as cube-map parameterization and parabolic parameterization [18, 19], which does reduce this limitation to some extent.

Shadow mapping has a number of useful advantages that have helped it become one of the more popular shadow algorithms in common use. As well as being implementable on common graphics hardware, its image-based nature means that it scales well with increasing occluder complexity compared to geometry-based algorithms such as shadow volumes (covered in Section 2.3). The fact that it doesn't impose any restrictions on the geometric construction of objects casting or receiving shadow has helped make it popular as this results in a low implementation burden. Also, adjusting the resolution of the shadow map offers a simple and direct control over the algorithm's quality and performance scaling, a property that makes it well-suited to real-time applications where hardware speed and scene complexity often vary considerably.

2.2.2 Percentage Closer Filtering

Percentage closer filtering is a method proposed by Reeves *et al.* [20] to reduce the visual problems associated with shadow map aliasing. In general, averaging of adjacent

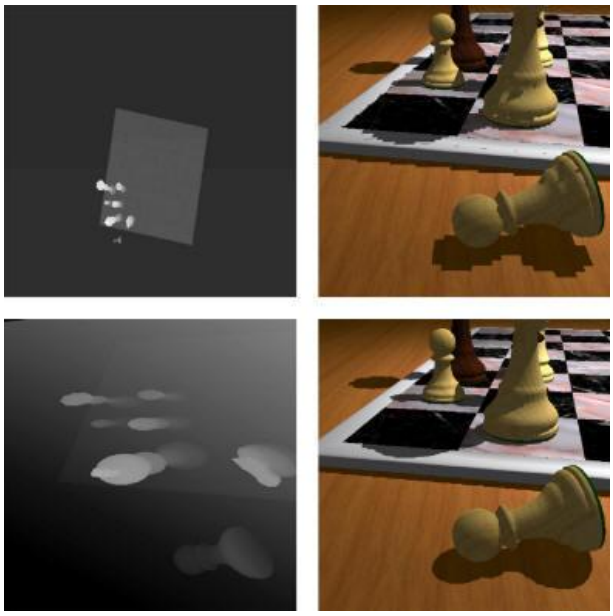

Figure 2.4: Shadow quality improvements of the perspective shadow map algorithm. The top images are of standard shadow mapping and have bad shadow map aliasing. The bottom images are of perspective shadow mapping and show a marked reduction in shadow map aliasing for the same shadow map resolution.

depth values in shadow maps will lead to incorrect shadowing, and percentage closer filtering gets around this by instead averaging the boolean comparison results within the extent of a 2D filter kernel [20]. A typical filter kernel size is 3×3, resulting in nine shadow map lookups and comparisons in place of the original single lookup and comparison. The final shadowing term for a fragment is now given as a value in the range $0 \rightarrow 1$ and is the number of comparisons that had $A < B$, divided by 9. The visual result of this filtering is illustrated in Figure 2.3. There is a blurring of shadow map aliasing artifacts along shadow boundaries that can also be seen as a crude early approximation of soft shadowing, although there is no notion of an area light source. Increasing the extent of the filter kernel will increase the blurring effect and further soften the shadow edges, however larger kernels require many more shadow map lookups and comparisons that place a significant burden on available video memory bandwidth.

2.2.3 Perspective Shadow Maps

Perspective shadow mapping tries to reduce the visual artifacts caused by shadow map aliasing and insufficient shadow map resolution by allocating more shadow map resolution to those objects that are closer to the viewer [21]. It does this by minimizing the effect of undersampling caused by $d_s r_s / r_i$ tending towards zero, which is achieved by keeping r_s / r_i close to constant.

Perspective shadow maps are computed in normalized device coordinates [22], that is, following perspective division. Both the scene and the light source are transformed

by the camera matrix and a shadow map is then generated in much the same way as in the standard shadow mapping algorithm. However, because the shadow map now sees the scene after perspective projection has occurred the amount of perspective aliasing is significantly reduced in some cases. Figure 2.4 demonstrates the improvement that perspective shadow mapping can have on shadow quality.

Unfortunately, in many common scenarios perspective shadow maps collapse down to the same result as the original shadow mapping algorithm and provide no increase in quality whatsoever. This is particularly common when using point light sources that have a large depth range in post-projective space [21], a common property of many point lights. The problem of self-occlusion is also increased by perspective shadow maps, the bias introduced to solve the problem in standard shadow mapping does not work once the non-uniform object scaling into post-projective space has occurred, and a constant depth offset in shadow map space must be used instead, which may require custom adjustment for different scenes.

2.3 Shadow Volumes

The shadow volume algorithm approaches shadowing from a purely geometric standpoint and is therefore classified as a geometry-based algorithm. The approach was first described by Crow in 1977 [23] and then implemented by Heidmann in 1991 [24].

The algorithm finds the silhouette of all occluders from the position of the light source, then extrudes this silhouette away from the light source thus forming a *shadow volume*. Objects outside this volume are illuminated and objects inside are in shadow.

Shadow volume construction is a two-step process. The first step is to find the silhouette of the occluder as viewed from the position of the light source. Computing the silhouettes exactly would be prohibitively expensive, so instead we find *possible silhouette edges*. The standard method defines possible silhouette edges as the subset of all edges in the occluder that are shared by a triangle facing towards the light and a triangle facing away from the light. This typically gives a superset of the true silhouette but is sufficient for the algorithm to work. Some implementations require that the occluder geometry be 2-manifold, that is, each edge in the occluder be shared by exactly two triangles, which speeds up silhouette determination as each edge can then be assumed to be used by exactly 2 triangles.

Once the set of possible silhouette edges for an occluder has been found these edges must then be extruded away from the light source to create the shadow volume. This edge extrusion is illustrated in Figure 2.5. Each edge results in the generation of a quadrilateral made of the two edge points and the two edge points projected away from the light source. The edge points are typically projected to infinity by setting the homogeneous w coordinate to zero as described in [25].

For each fragment in the rendered image we count the number of shadow volume quadrilaterals that are crossed between the view point and the fragment. Front-facing quadrilaterals increment the count, and back-facing shadow volumes decrement the count. If the final count is greater than zero then the point is in shadow, otherwise it is illuminated.

Hardware acceleration of shadow volumes is possible using the stencil buffer [24, 26]. The stencil buffer is a 2D array of 8-bit unsigned integers that has the same resolution as the final rendered image. First the final depth map for the scene is rendered

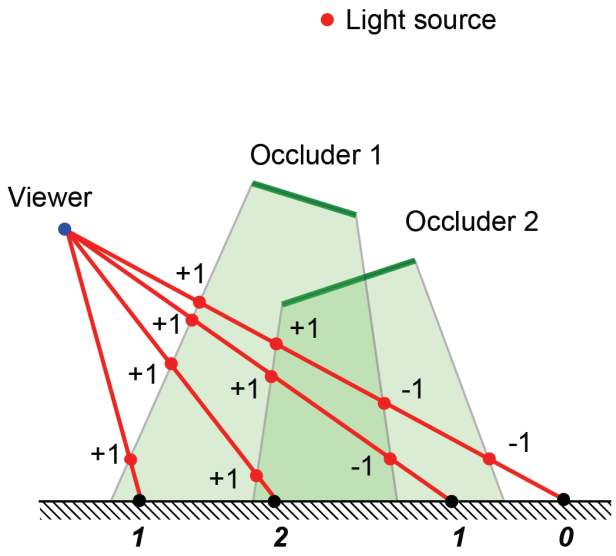

Figure 2.5: How the shadow volume algorithm determines shadows. The count is incremented for front facing shadow quadrilaterals and decremented for back facing ones. A final count of zero indicates an illuminated fragment.

so that the depth values at each pixel indicate the closest visible surfaces. Second, the shadow volume polygons are rendered such that fragments on their front faces increment the corresponding value in the stencil buffer and fragments on their back faces decrement the corresponding value. This process of incrementing and decrementing is illustrated in Figure 2.5 and described by the pseudo-code in Listing 2.2.

Listing 2.2: Pseudo-code for shadow volume geometry determination and stencil buffer rendering.

```
for all shadow casting objects
    compute potential silhouette edges
    compute shadow volume quads from the light
        position and the silhouette edges
end for

for all front-facing shadow quads from viewpoint
    if Z-buffer test passes then
        increment stencil buffer value
    end if
end for

for all back-facing shadow quads from viewpoint
    if Z-buffer test passes then
        decrement stencil buffer value
    end if
end for
```

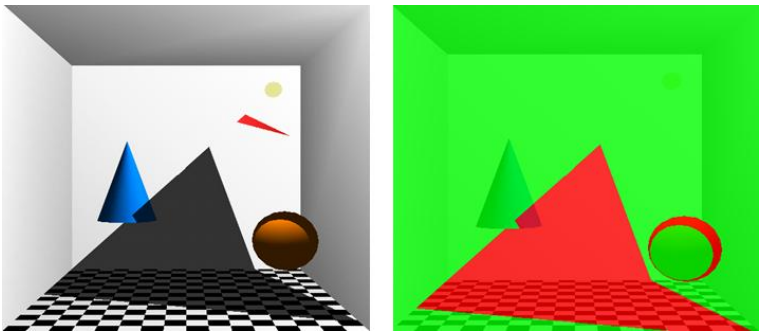

Figure 2.6: Stencil shadow volumes. The left image is a final shadowed scene lit by a single light source. The right image visualizes the contents of the corresponding stencil buffer, where green is a stencil value of 0 and red is a stencil value of 1.

The final scene can now be rendered using a test against the values in the stencil buffer to determine shadowing. This is a per-fragment comparison of the integer value in the stencil buffer and is accelerated by all common graphics processors and libraries. When the stencil buffer is used in this manner the algorithm is sometimes referred to as *stenciled shadow volumes*.

2.3.1 Evaluation

The result of the per-fragment stencil comparison is binary, resulting in pixel-exact hard shadow edges that have no penumbra, as illustrated in Figure 2.6.

The speed of the shadow volume algorithm is strongly linked to the number of possible silhouette edges that are extruded, as well as the number of pixels the resulting shadow volume quadrilaterals cover. This means that performance can drop significantly on complex meshes that generate a large number of possible silhouette edges, meaning that shadow volumes often don't scale up well to use with high polygon meshes. The classic example of an object that will perform poorly with shadow volumes is a chain link fence with a light shining through it. Each link will result in several shadow volume quadrilaterals and casting shadows through the whole fence will generate a huge amount of shadow volume geometry that will slow the algorithm down considerably. Implementation details for improving shadow volume performance are discussed in [27].

Another problem with the standard shadow volume algorithm is that it only works when the viewpoint is not inside one of the shadow volumes. When this occurs the stencil counting breaks down leading to the inversion of the shadowing result. That is, fragments that should be in shadow become illuminated and fragments that should be illuminated become shadowed. The solution to this is the *z-fail* variation that is discussed in Section 2.3.2.

The final drawback of shadow volumes is that they require a lot of detailed information about mesh structure in order to compute object silhouettes. Adjacency data structures need to be created in a pre-process to determine an object's inter-triangle connectivity, that is, where triangles share edges. This requires a degree of mesh prepa-

ration that the shadow mapping algorithm avoids.

Shadow volumes also have a number of advantages that make them suitable to real-time applications. They are fully hardware accelerated on most graphics processors, work particularly well for omnidirectional point light sources, and are view-independent.

2.3.2 The Z-fail Algorithm

Everitt and Kilgard [25] describe a solution to the problem of incorrect shadow determination when the viewpoint is contained inside a shadow volume. Their approach is based on the observation that counting the visible front and back faces of shadow quadrilaterals between the near plane of the camera and the surface yields the same result as counting the *hidden* front and back faces between the surface and the *far* plane. Increments and decrements now occur when a fragment from a shadow volume quadrilateral *fails* the depth test, as opposed to when it passes the depth test as in the original algorithm. This shifts the problem from clipping caused by the near plane to the clipping caused by the far plane [25]. However, far plane clipping can be avoided when rendering a perspective view by using a custom projection matrix that pushes the far plane out to infinity, resulting in robust stencil shadow volumes in all circumstances. The standard perspective projection matrix P is given by

$$P = \begin{bmatrix} \frac{2\times Near}{Right-Left} & 0 & \frac{Right+Left}{Right-Left} & 0 \\ 0 & \frac{2\times Near}{Top-Bottom} & \frac{Top+Bottom}{Top-Bottom} & 0 \\ 0 & 0 & -\frac{Far+Near}{Far-Near} & -\frac{2\times Far\times Near}{Far-Near} \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{2.2}$$

Note that only the third row depends on the far plane distance. As $Far \to \infty$ the matrix becomes

$$P_{inf} = \lim_{Far\to\infty} P = \begin{bmatrix} \frac{2\times Near}{Right-Left} & 0 & \frac{Right+Left}{Right-Left} & 0 \\ 0 & \frac{2\times Near}{Top-Bottom} & \frac{Top+Bottom}{Top-Bottom} & 0 \\ 0 & 0 & -1 & -2\times Near \\ 0 & 0 & -1 & 0 \end{bmatrix} \tag{2.3}$$

Using this projection matrix eliminates far plane clipping. One question this raises is what the effect will be on depth buffer precision, which now has to cover an infinite z range. Fortunately, due to the projective nature of perspective transformation a minimal amount of depth buffer precision is wasted by an infinite far clipping plane. Compared to a finite far clipping plane, depth precision is compressed by a factor of

$$\frac{Far-Near}{Far} \tag{2.4}$$

In typical usage scenarios such as $Near = 1.0$ and $Far = 100.0$ this results in a 1% reduction in depth precision. Because depth precision itself is non-uniform due to perspective division, with areas closer to the eye being given more precision, the loss of precision is shifted more towards the far clip plane with the result that the slight loss of precision is negligible in practice. It should also be noticed that rendering of z-fail

Figure 2.7: Simulating soft shadows with jittered shadow volumes. A cluster of 12 low intensity lights is used to approximate an area light source.

shadow volumes requires that both ends of the shadow volume be capped in order to enclose the volume [25].

Z-fail shadow volumes make the shadow volume algorithm robust enough for use in arbitrary scenes with no restrictions on the location of the viewpoint, and because of this they have been used in many commercial real-time graphics applications.

2.3.3 Jittered Shadow Volumes

Although the standard shadow volume algorithm is concerned with the generation of hard shadows, it is possible to adapt the algorithm to generate soft shadows using a technique called jittered shadow volumes. This technique approximates an area light source by using a set of low intensity point lights clustered around the area light source. The contribution of each light is summed to get the final illumination result that bears a reasonable resemblance to physically correct soft shadows as illustrated in Figure 2.7.

Taking the accumulation of hard shadows from a variable number of sample points on the area the light source is the most straightforward method to achieve soft shadowing. The main problem with this is the number of samples required to produce acceptable shadows, which precludes use in real-time applications. If the shadow penumbrae are large, either because the shadow casting object is close to the light source or a large area light source is in use, then a lot more low intensity light sources will be required to avoid banding artifacts. These performance issues generally means that other soft shadowing algorithms such as those described in Chapter 3 perform considerably better than this naïve approach.

2.4 Hybrid Algorithms

Shadow mapping and shadow volumes each have different strengths and weaknesses. In light of this, a hybrid approach employing both techniques was proposed in 2000 by McCool [28] that uses a true silhouette to perform shadow volume rendering. This true silhouette is found by applying edge detection algorithms to a depth buffer rendered from the light's point of view. McCool's implementation used the CPU to compute the silhouette edges which meant reading the depth buffer back from the graphics processor every frame. A more modern implementation that performs this step using shaders on the GPU may now be possible, depending on the edge detection algorithms in use.

This approach produces surprisingly good shadowing results in simple scenes [28]. The shadows are hard due to the final shadowing being done with shadow volumes, but the scalability issues that occur with shadow volumes as mesh complexity increases are reduced because only the true silhouette is extruded rather than all of the possible silhouette edges. Possible silhouette edges are typically far more numerous than the edges that make up the true silhouette.

Chapter 3

Soft Shadow Algorithms

We now shift focus to algorithms that extend the shadowing techniques covered in Chapter 2 in order to produce real-time soft shadows. The optical basis for the presence of umbrae and penumbrae in real world shadows has already been discussed in Section 1.2. It is only in the last few years that dynamic shadowing has become commonplace in real-time computer graphics applications, and significant recent developments in hardware capability and performance has resulted in new techniques becoming possible.

In this chapter we describe two algorithms for real-time soft shadow generation. The first, percentage closer soft shadows [29], extends the shadow map algorithm from Section 2.2 and specifically the percentage closer filtering algorithm from Section 2.2.2 to achieve accurate modeling of the umbrae and penumbrae of area light sources. The second algorithm, penumbra wedge shadow volumes [2, 30, 31], is based on the shadow volume algorithm from Section 2.3 and achieves similar modeling of area light sources.

3.1 Percentage Closer Soft Shadows

The percentage closer soft shadows algorithm was first described by Fernando in 2005 [29] and is based on the observation that percentage closer filtering results in a blurring of the shadow edges. This naturally leads to the possibility of dynamically adjusting the size of the filter kernel based on a metric that indicates how soft the shadow should be at a given fragment. Recent advancements in shader programming have made this possible to implement in real-time [32]. Notably, the algorithm uses a standard shadow map and only changes the per-fragment shadow determination process, making it pleasingly simple to merge into existing shadowing frameworks.

As mentioned in Section 2.2.2 and Figure 2.3, on its own percentage closer filtering gives a crude appearance of soft shadows. However, because the filter size is fixed the amount of smoothing applied is *uniform*. This means that shadow edge softness does not vary, resulting in physically inaccurate shadowing that would be more correctly termed *blurred shadows* than soft shadows.

The fundamental problem for the percentage closer soft shadow algorithm is determining the filter kernel size to use for a given fragment. As illustrated in Figure

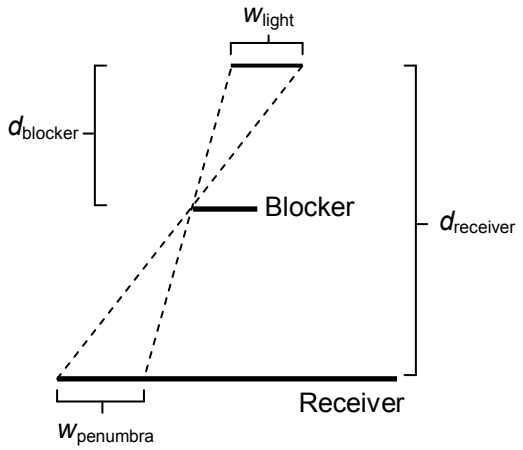

Figure 3.1: The relationship between blocker depth, receiver depth, light size, and fragment penumbra size.

3.1, a good value for kernel size is the size of the light projected through the blocking object that is casting the shadow. The variables that affect the calculation of the kernel size $w_{penumbra}$ are the size of the light source, w_{light}, the distance from the light source to the blocker, $d_{blocker}$, and the distance from the light source to the receiver, $d_{receiver}$. Applying the elementary geometric rule of similar triangles yields Equation 3.1.

$$w_{penumbra} = \frac{(d_{receiver} - d_{blocker}) \times w_{light}}{d_{blocker}} \tag{3.1}$$

The light source size w_{light} is a constant value for the light. $d_{receiver}$ is also readily available as it is used by the standard shadow map algorithm to do the depth comparison (it is the B value). The only piece of Equation 3.1 which is currently not available is $d_{blocker}$, and so it needs to be calculated for each fragment. The process of shadow determination in percentage closer soft shadows can be broken down into the following three stages, all of which are performed for every rasterized fragment.

1. **Blocker search**: determine $d_{blocker}$ by searching the shadow map.
2. **Penumbra size estimation**: estimate the penumbra size using Equation 3.1.
3. **Variable filtering**: perform a series of filtered shadow map depth comparisons with an appropriately sized 2D filter kernel.

We now address each of these steps in detail.

3.1.1 Blocker Search

Blocker search is the process of searching the shadow map to find the average blocker distance for a fragment. Instead of directly looking up the fragment's light space depth value from the shadow map we define a rectangular search region in the shadow map

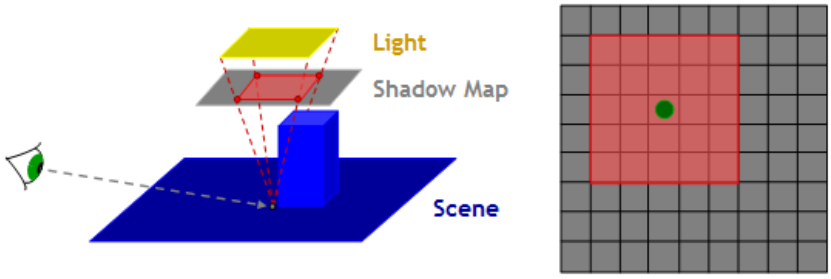

Figure 3.2: The blocker search region used in percentage closer soft shadows. The left image shows how the search region size is based on light source size and the distance from the light to the fragment. The right image shows the search region in the shadow map.

centered around the shadow map pixel that would be used for the depth comparison in standard shadow mapping. The size of this search region depends on the distance from the fragment to the light source, $d_{receiver}$, and the size of the light source, w_{light}. This relationship is illustrated in Figure 3.2.

Once the region of the shadow map to search in has been determined we iterate over all the depth values in this region and use them to compute the average blocker distance. This is achieved by taking the average of all the depth values that are less than $d_{receiver}$. Restricting the average to the subset of values that are $< d_{receiver}$ cuts out objects that are further away from the light source than the receiving object. The number of samples taken in the blocker region has a significant impact on shadow quality and algorithm performance. Testing on real-world scenes found that a 6×6 sampling grid gives acceptable results in most cases. Adaptively adjusting the density of the search grid based on the area covered in the shadow map is a potential improvement that could be made. Pseudo-code for the blocker search process is given in Listing 3.1.

Listing 3.1: Pseudo-code for blocker search in the percentage closer soft shadow algorithm.

```
// Loop through search region summing depth values
for i = 1 to sample count
    for j = 1 to sample count
        sampleDepth = shadow map value at location
                      (i, j) in the search region

        if sampleDepth < receiverDepth then
            // Found a blocker in the shadow map
            blockerSum += sampleDepth
            blockerCount++
        end if
    end for
end for

return blockerSum / blockerCount
```

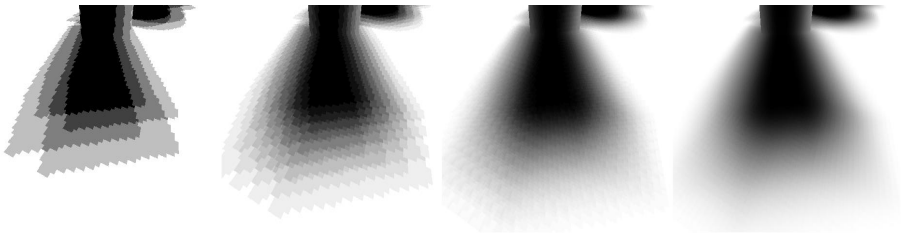

Figure 3.3: The effect of filter kernel size on percentage closer soft shadow quality. Moving from left to right the kernel sizes are 2×2, 4×4, 8×8 and 16×16. Notice how the penumbra realistically enlarges and softens.

It should be noted that the presence of a conditional statement in a fragment processing code requires graphics hardware capable of fragment shader branching. Such hardware was first made available in 2004 by NVIDIA's GeForce 6 series of graphics processors [33].

3.1.2 Penumbra Size Estimation

Once a value for $d_{blocker}$ has been calculated, Equation 3.1 can be used to approximate the width of the light's penumbra, $w_{penumbra}$, at the current fragment.

This only provides an approximation of penumbra size, however the results are typically satisfactory for most applications as many phenomena are modeled qualitatively by this approach, including penumbra size, umbra size, as well as the total disappearance of the umbra in the case of large area light sources.

3.1.3 Variable Filtering

The third and final step is to apply a 2D percentage closer filter to determine the final shadowing term. The size of this filter will vary from fragment to fragment based on the estimated penumbra size. A standard 2D percentage closer filtering implementation is given in Listing 3.2.

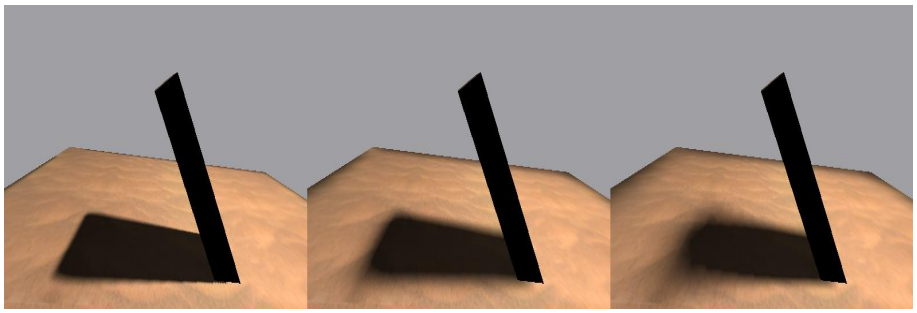

Figure 3.4: Percentage closer soft shadows with different size area light sources. The size of the area light source increases moving from left to right. Notice how in the right-hand image the shadow starts out sharp at the base of the occluder and quickly becomes very soft. This example is using an 8×8 filter kernel and a 6×6 blocker search sampling grid.

Listing 3.2: Pseudo-code for percentage closer filtering in the percentage closer soft shadow algorithm.

```
// Loop through search region summing depth values
for i = 1 to sample count
    for j = 1 to sample count
        sampleDepth = shadow map value at location
                      (i, j) in the search region

        if sampleDepth < receiverDepth then
            // Found a blocker
            blockerSum += sampleDepth
            blockerCount++
        end if
    end for
end for

return blockerSum / blockerCount
```

The number of samples taken has a significant impact on both visual quality and performance. The visual effect of different fixed size sample kernels is illustrated in Figure 3.3. As expected, kernels with a higher sample density show significantly better shadowing results, particularly when the penumbra size is large.

The observation that more samples are needed to obtain high quality soft shadows leads to the possibility of adaptively adjusting the filter kernel for each fragment. Clearly not all fragments need a 16×16 filter kernel, only those that are part of a large penumbra will benefit noticeably from this level of filtering.

3.1.4 Summary

Provided the sample density of the blocker search and the variable filtering operations are sufficient the result of the percentage closer soft shadow algorithm is visually pleasing, as illustrated in Figure 3.4. Figure 3.3 illustrates the types of artifacts that occur when sample density is not high enough, although in most real-world scenes a certain level of aliasing is acceptable as it is often hidden by surface textures or other effects. Figure 3.4 also demonstrates the algorithm's ability to adjust the size of the area light source and have the shadowing result react in a realistic manner. However, as was mentioned in Section 3.1.3, large area light sources result in large penumbrae which in turn require larger filter kernels to produce smooth shadow gradients in the final image. This means that the texture memory bandwidth required grows rapidly with light source size, which generally restricts the use of large area lights in real-time applications.

Percentage closer soft shadows offer perceptually correct modeling of umbrae and penumbrae of area light sources and are easily worked into existing graphics pipelines that support shadow mapping. The blocker search sampling grid and percentage closer filter kernel size offer straightforward methods of adjusting the algorithm's performance and quality scaling, making it a suitable technique for real-time use.

3.2 Penumbra Wedge Shadows

We now look at the recently developed geometry-based *penumbra wedge* algorithm for soft shadow generation that is based on Crow's [23] shadow volume algorithm described in Section 2.3. It was originally proposed by Akenine-Moller and Assarsson in a series of papers published from 2002 to 2004 [2, 30, 31], and has since been extended by other researchers [6]. Due to similarities with the original shadow volume algorithm it is sometimes referred to as the *soft shadow volume algorithm*, but we feel this is too general a name and so refer to it as the penumbra wedge algorithm.

The penumbra wedge algorithm builds up a *visibility buffer* which will contain the final per-pixel shadowing terms. This buffer is usually the same resolution as the final image, although the resolution can be reduced to achieve performance and quality scaling. Visibility buffer construction is broken down into the following three steps.

1. Estimate the umbra
2. Build penumbra wedges
3. Rasterize penumbra wedges to correct the umbra and add penumbrae

We now analyze each of these steps in detail, and then discuss issues related to a hardware implementation.

3.2.1 Umbra Estimation

The first step in the penumbra wedge algorithm is to generate hard shadows using the stencil shadow volume algorithm described in Section 2.3. The resulting hard shadows are taken as an *overestimation* of the umbra required for correct shadowing, and are

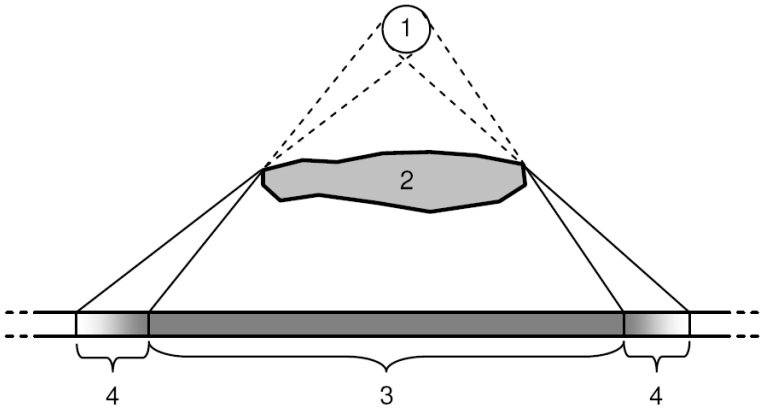

Figure 3.5: Penumbra wedge extrusion. This illustration uses a spherical area light source (1), which is extruding two penumbra wedges from the object (2). The umbra of the shadow (3) is correctly defined and illumination is varied through the penumbra wedges (4).

rendered into the visibility buffer. These hard shadows are corrected by the remaining steps in the algorithm, that is, the umbra is reduced in size and penumbrae are added. This is achieved using new primitives called *penumbra wedges*.

3.2.2 Wedge Construction

Penumbra wedge construction proceeds as follows. First, object silhouettes as seen from the light source's point of view are computed using the methods described in Section 2.3. However, instead of extruding a single quadrilateral we build a wedge for each silhouette edge that encloses the penumbra area cast by the edge. This wedge may overestimate the size of the penumbra, but this does not cause problems as will be explained shortly. The size of the extruded wedge is determined by the size of the light source and the distance from the light source to the edge, as illustrated in Figure 3.5; larger light sources will lead to larger penumbra wedges.

It should be noted that while this technique simulates the penumbrae of an area light source, it still uses a single point for silhouette determination. For spherical light sources this would usually be the center of the light sphere, for rectangular light sources the center of the rectangle, and similarly for other shapes. While this is a reasonable approximation for small to moderate light sources, it can result in incorrect shadowing for large area light sources. This problem is covered in more detail in Section 3.2.5.

For an arbitrary light source, the precise penumbra volume for a silhouette edge is the volume of an infinite cone swept from one vertex to the other, where the cone is determined by reflecting the shape of the light source through the edge. This is illustrated in Figure 3.6 for both spherical and rectangular light sources.

Precisely computing these swept volumes in real-time is not possible, and so the penumbra wedges are constructed in such a way as to robustly *enclose* this swept volume. Penumbra wedges are defined by front, back, left and right planes that enclose their volume. For a given silhouette edge with vertices e_0 and e_1, its wedge is computed

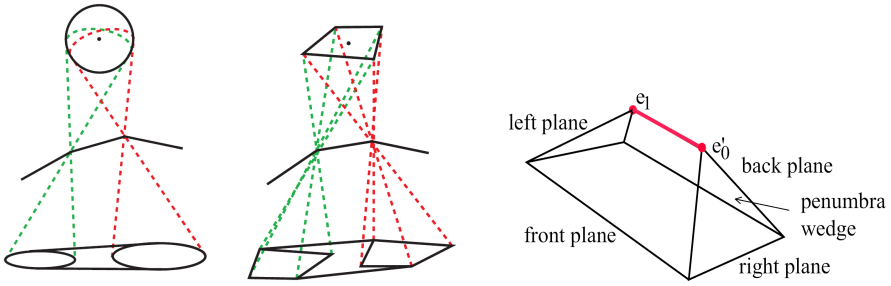

Figure 3.6: Visualization of penumbra wedges. The left two images show the exact penumbra volume as the swept volume of the light source projected through the edge for spherical and rectangular light sources. The right image shows the four planes that enclose the penumbra wedge for the edge (e'_0, e_1).

as follows. We first determine which of e_0 or e_1 is closer to the light source; assume, without loss of generality, that this is e_1. The other vertex, e_0, is now moved towards the light source center, l_c, until it is the same distance from the light as e_1. We call this new vertex e'_0, and together with e_1 it forms the top of the penumbra wedge. This new edge ensures that the computed wedge will contain the entire penumbra volume of the original edge. Section 3.2.3 explains why this overestimation of the actual penumbra volume does not cause visual artifacts.

With reference to the right hand image in Figure 3.6 it can be seen that the front and back planes both contain the edge (e'_0, e_1) and are rotated around the edge such that they barely touch the light source on either side. The right plane contains both the point e'_0 and the vector that is perpendicular to both the edge vector, $e_1 - e'_0$, and the vector from e'_0 to the light source, $l_c - e'_0$. Similarly, the left plane contains both the point e_1 and the vector that is perpendicular to both the edge vector, $e_0 - e'_1$, and the vector from e_1 to the light source, $l_c - e_1$. Like the front and back planes, the left left and right planes are rotated such that they barely touch the outside of the light source.

If a silhouette edge's vertices are significantly different distances from the light source then the computed penumbra wedge will not be a good fit for the actual penumbra volume, resulting in a large number of unnecessary fragments being generated when rasterizing the wedge. This problem is covered in more detail in Section 3.2.5.

Finally, the vertices needed to render the penumbra wedges are found by intersecting the bounding planes. Specific implementation details for wedge generation are covered in Section 3.2.4.

3.2.3 Visibility Computation

The next step is to use the penumbra wedges to alter the hard shadows that are present in the visibility buffer. Each penumbra wedge is rendered, and for each rasterized fragment that falls inside a penumbra we compute the fraction of the light source that is occluded by the original edge. That is, if there was a viewer at the fragment position looking at the light source, we are calculating the percentage of the light source that would be occluded by the *original edge* that generated this wedge.

Before describing how this visibility term is calculated it is helpful to look at how

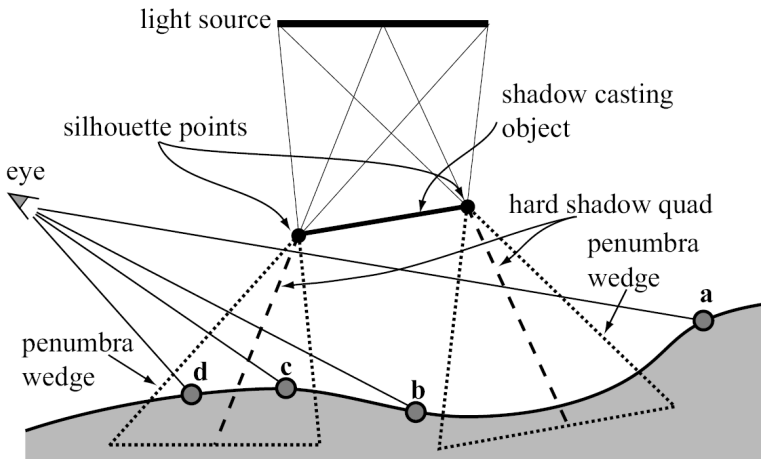

Figure 3.7: Fragment light visibility computation in the penumbra wedge algorithm. The hard shadow quads used to overestimate the umbra are shown as dashed lines. Penumbra wedges are outlined with dots.

it will be used. Assuming we have computed per-fragment light visibility, we will either add or subtract this value from the visibility buffer. Recalling that the visibility buffer initially contains the hard shadows that are an overestimation of the umbra, if the fragment falls *inside* this umbra region then we will add its light visibility fraction to the visibility buffer. Similarly, if the fragment falls *outside* this umbra region then we will subtract its light visibility from the visibility buffer. This will result in a reduction in the size of the umbra and the addition of a proper penumbra. Figure 3.7 illustrates how this process works for example fragments a, b, c and d. Fragments a and b lie outside penumbra wedges and so their shadowing result is the same as with the original shadow volume algorithm. Fragments c and d are inside the left wedge and so light source visibility is calculated for them and used to adjust the corresponding values in the visibility buffer.

Given a single penumbra wedge generated by the possible silhouette edge $e = (e_0, e_1)$ and a point p inside that wedge, the visibility of the light source from p with respect to the edge e is calculated as follows. The semi-infinite hard shadow quadrilateral Q that contains the edge e is projected onto the light source as seen from position p. This is illustrated for a rectangular light source in the left half of Figure 3.8. Note that in the case where one of the edge vertices does not lie between p and the light plane it must first be clipped to a near plane close to p that has the same normal as the light plane. The projection of Q onto the light source results in a second semi-infinite quadrilateral that is defined by the projected edge endpoints and two infinite edges, parallel with the vectors from the light source center to each edge endpoint, extended outwards from the light source center.

Once the semi-infinite hard shadow quadrilateral Q has been projected onto the light source, we need to calculate the fraction of the light source that is covered by this projection (illustrated by the dark shaded area in Figure 3.8). This is the area of the intersection of the light source and the projection of Q divided by the total area of the light source, and can be solved using standard 2D geometric clipping algorithms.

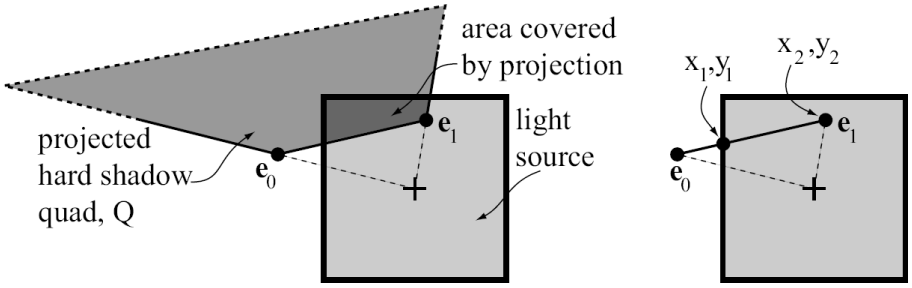

Figure 3.8: Determination of light source visibility in relation to edge $e = (e_0, e_1)$. The left image shows the projection of the hard shadow quad Q onto a rectangular light source. The coverage area is the area of the light source covered by the projection of Q. The right image shows the clipping of projected edge e to the rectangular light source, the clipped (x_1, y_1) and (x_2, y_2) coordinates are used to speed up coverage computation.

This fraction of the light source covered by the projection of Q is called the *coverage*. Note that the computed coverage value for a given fragment will always be in the range $0 \rightarrow 0.5$.

After the final coverage value for a fragment has been calculated it is either added or subtracted from the visibility buffer depending on whether it lies behind or in front of Q. This assumes that the normal of hard shadow quadrilaterals point out from the umbra, if they faced into the umbra then fragments in front of Q would need to be added to the visibility buffer and those behind Q subtracted from the visibility buffer.

Following rasterization of all penumbra wedges the visibility buffer will contain the final shadowing term for each pixel. This is usually multiplied with a per-pixel lighting term, surface texturing terms, and then added to an ambient lighting term to produce final per-pixel illumination values.

3.2.4 Implementation Details

Implementing the penumbra wedge algorithm in hardware is significantly more complicated than the original shadow volume algorithm. It requires accurate wedge geometry calculation and complex per-fragment computation to calculate light visibility. Fortunately, current generation hardware is sufficiently versatile and has enough processing power to allow real-time rendering of soft shadows using the algorithm. Two examples of soft shadows created by the penumbra wedge algorithm are shown in Figure 3.9. We now discuss some key implementation details.

Wedge Fragment Determination

One problem that has not yet been addressed is how to determine if a given fragment generated by the rasterization of a penumbra wedge is in fact part of a penumbra receiving surface. Without this information we do not know whether to compute visibility and update the visibility buffer for this fragment. The correct scene depth position for a rasterized penumbra fragment is read from a depth buffer of the scene that is generated in a separate pass prior to shadow rendering, giving a 3D position for the fragment.

Figure 3.9: Soft shadows created by the penumbra wedge algorithm. Note the sharp shadows where the objects contact the floor and the realistically enlarging penumbrae towards the tips of the shadows.

Now that we have a fragment position it can be used to determine whether the fragment lies inside the penumbra wedge volume. This is done by passing the planes that enclose the penumbra wedge to the fragment processor which then uses them to clip fragments that fall outside the penumbra volume. This explains why the extra fragments created by the penumbra wedge overestimation described in Section 3.2.2 do not have a significant impact on performance, as they are typically fragments that are quickly clipped away by the fragment processor and so don't go through the costly light visibility calculations described in Section 3.2.3.

For hardware implementations each penumbra wedge is split along its hard shadow quadrilateral Q into inner and outer halves. The two penumbra halves are rendered separately, splitting the addition and subtraction from the visibility buffer into two separate phases. This is an implementation detail that reduces the hardware requirements so the algorithm can be implemented on a wider array of graphics processors.

Coverage Optimization

As described in Section 3.2.3, to calculate light visibility for a point in a penumbra wedge the original edge is projected back onto the light source and extruded away from the light source center. Geometric clipping algorithms are then used to clip the projection of Q to the light source boundaries in order to compute visibility (Figure 3.8). However, these geometric clipping operations are costly to perform and an alternative solution that makes use of pre-computed lookup textures can be used speed up this step. Note again that we are dealing with rectangular light sources, spherical light sources simplify these computations significantly.

If the projected edge is clipped inside the bounds of the light source (illustrated by the right half of Figure 3.8), then the coverage area becomes a function of the clipped edge coordinates, (x_1, y_1) and (x_2, y_2). The goal now is to calculate the area of the sector subtended by the clipped edge. This area is equal to the total light source area between the two vectors connecting the center of the light to the clipped edge endpoints, minus the area of the triangle formed by the light source center and the two

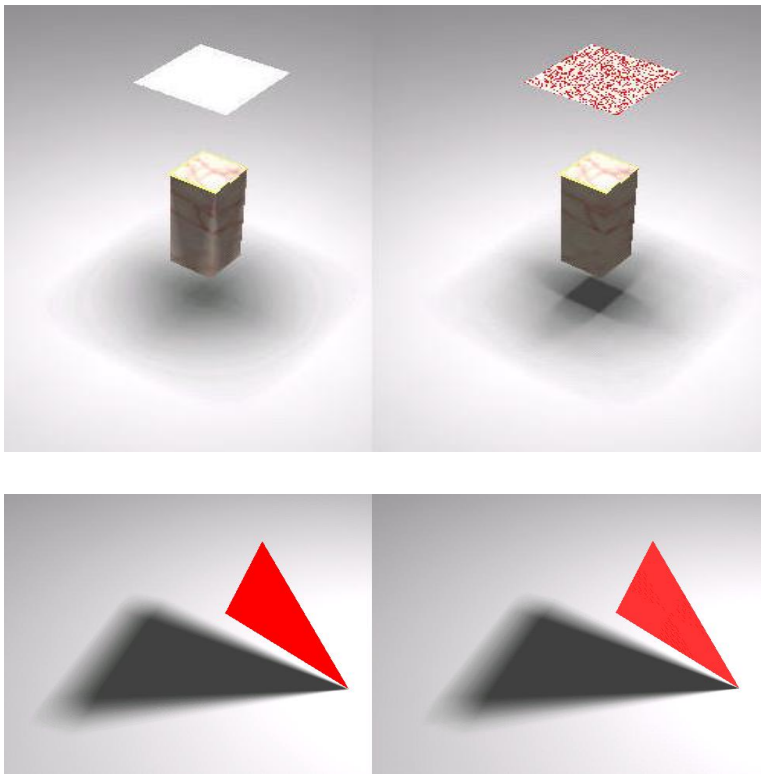

Figure 3.10: Comparing the penumbra wedge algorithm to reference images generated by taking 1024 hard shadow samples. The left hand images were generated by the penumbra wedge algorithm. The upper images shows the penumbra wedge algorithm diverging significantly from the reference image, whereas in the bottom images the results are extremely close.

end points. The former value is found by looking up into a pre-computed cube-map texture and the latter can be calculated directly. This method is significantly faster than naïve geometric clipping as it reduces the amount of per-fragment computation.

3.2.5 Summary

As demonstrated by Figures 3.9 and 3.10, the penumbra wedge algorithm results in visually pleasing soft shadows that do an excellent job of modeling real-world shadow phenomena. Both umbrae and penumbrae are simulated and in some cases the results of penumbra wedge rendering are actually physically correct, that is, they are very close to the results from 1024 hard shadow samples (see the lower images in Figure 3.10). While the algorithm is suitable for real-time applications, provided reasonably modern graphics hardware is present, it does still have a number of visual artifacts.

One such artifact stems from the algorithm's use of a single point for silhouette determination. For small area light source this approximation introduces only a small amount of error, but for larger area light sources such as the one in the top-left image

of Figure 3.10 the error introduced can be significant. The red dots in the top-right image of Figure 3.10 are the sample points on the area light source used, and the object silhouette is not the same for all of them. The penumbra wedge algorithm ignores this and so produces physically incorrect shadows, although they are arguably still visually acceptable in most cases.

Runtime performance and quality scaling can be achieved in the penumbra wedge algorithm by altering the size of the visibility buffer. Reducing the resolution of this buffer will diminish the amount fragment processing load and also reduce shadow quality due to the $1 : 1$ mapping from visibility buffer pixels to screen pixels being lost.

Finally, the penumbra wedge algorithm inherits the advantages of standard shadow volumes such as view independence and natural support for omnidirectional light sources, as well as their disadvantages such as performance issues when used with high resolution meshes and requiring triangle connectivity information in order to determine possible silhouette edges.

Chapter 4

Geometry Shaders

Having covered algorithms for real-time soft shadow generation, we now present new methods for improving their implementation quality and performance by using new hardware features made available by the latest generation of graphics processors. Specifically, we focus on how to achieve this using the new *geometry shader* feature that allows the creation of geometric primitives directly on the GPU, something that has not previously been possible. We show how this feature can be used to accelerate and simplify shadow volume rendering, and then extend this technique to the penumbra wedge soft shadow algorithm.

As of November 2007, the only consumer-level graphics processors that support geometry shading are the GeForce 8 series from NVIDIA and the Radeon HD 2000 series from ATI. The GeForce 8 series from NVIDIA was the first of these to be released, with initial availability in November 2006.

4.1 Programmable Pipeline

Prior to the introduction of geometry shaders the graphics pipeline had two major programmable stages, vertex processing and fragment processing. These allow custom shader programs written in a C-like language to be executed for every vertex in a mesh and for every pixel that is filled in the final image. Vertex shaders allow alteration of vertex properties such as position, color, and normals. Similarly, fragment processing allows alteration of the fragment properties such as color. These programmable stages have made it possible to implement many new effects with full hardware acceleration. Further coverage of topics relating to shaders and the programmable pipeline for the OpenGL and DirectX APIs can be found in [32] and [34].

4.2 Capabilities

The geometry shader is a new programmable stage in the graphics pipeline that is positioned between the vertex shader and the fragment shader. It allows the user to define custom programs that are executed for each geometric primitive that is submitted to the GPU. These programs are able to both alter incoming primitives and to create new ones.

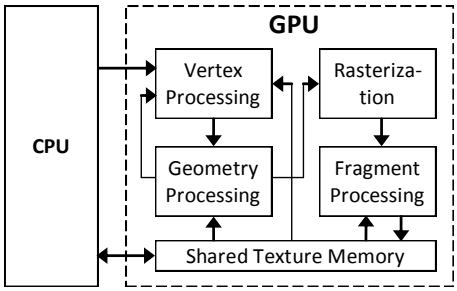

Figure 4.1: Visualization of data flow between the GPU's major pipeline stages.

This processing is all hardware accelerated on the graphics processor and requires no intervention from the CPU other than to specify what shader program to execute for incoming primitives. This eliminates the previous primitive assembly model of "one primitive in, one primitive out". A detailed technical specification of geometry shader functionality in OpenGL can be found in [35].

Geometry shading allows developers to dynamically adjust and create geometry on the GPU and gives a more global view of the mesh structure being processed. This can be used for animation, mesh tessellation, fur rendering, isosurface generation, procedural geometry, particle systems, and, as we will show, shadowing. Figure 4.1 illustrates the data flow between the programmable processing stages of a modern GPU.

A simple example of a geometry shader written in GLSL [32] is given in Listing 4.1. For a triangle mesh, the `main()` routine will be executed for every triangle in the mesh and `gl_VerticesIn`, the number of vertices in the input primitive, will always be equal to 3. Note that the input primitive type can also potentially be lines or points, so `gl_VerticesIn` is not always 3. This geometry shader doesn't do anything particularly useful, it simply passes through each vertex's position without altering it. The vertices seen by the geometry shader have already been processed by the vertex shader.

Listing 4.1: A simple GLSL geometry shader.

```
void main()
{
    for (int i = 0; i < gl_VerticesIn; i++)
    {
        // Copy vertex position
        gl_Position = gl_PositionIn[i];

        // Output a vertex
        EmitVertex();
    }
}
```

Before we can detail how geometry shaders are able to be used in shadow rendering it is necessary to describe the closely related new feature of adjacency primitives.

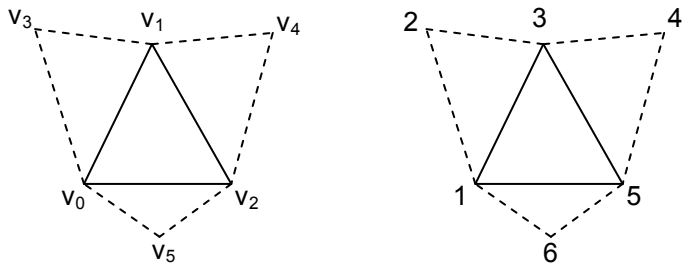

Figure 4.2: Including triangle adjacency information in an index array. For the vertex array $V_{array} = \{v_0, v_1, v_2, v_3, v_4, v_5\}$ the standard indices for the triangle (v_0, v_1, v_2) would be $(0, 1, 2)$. The right hand image gives the ordering of vertex indices for including adjacency information for the outlined triangle, the new index array with adjacency information for the triangle (v_0, v_1, v_2) is therefore $(0, 3, 1, 4, 2, 5)$.

4.3 Adjacency Primitives

Normal geometry is submitted to the GPU using a vertex array which has one entry per vertex, and an index array where each index is into the vertex array. This requires 3 indices per triangle in the simplest case. For example, with the vertex array $V_{array} = \{(0,0),(0,5),(5,0),(5,5))\}$ the index array required to make a square from these vertices out of two triangles would be $I_{array} = \{0, 1, 2, 2, 1, 3\}$. This primitive setup is called *indexed triangles*. Triangle strips are a commonly used index array representation that is more compact than indexed triangles in most cases, but for the purposes of this discussion they do not need to be considered.

Using this format for vertex and index data, each triangle specified by the index array is independent and has no knowledge of the surrounding mesh topology. Adjacency primitives change this by adding a new primitive type: *indexed triangles with adjacency*. When using this primitive type, triangles are no longer specified in isolation, and information about adjacent triangles is included in the index array. The indexed triangles with adjacency primitive type uses 6 indices per triangle instead of the original 3. The extra indices are used to indicate adjacent vertices, and this adjacency information is then made available to the geometry shader. The right half of Figure 4.2 illustrates the ordering of indices for a triangle when including adjacency information. For the triangle (v_0, v_1, v_2), instead of an index array of $\{0, 1, 2\}$, extra indices are inserted for the vertices adjacent to each triangle edge. The indices for this triangle including adjacency information is therefore $(0, 3, 1, 4, 2, 5)$.

The same basic principle can be applied in order to embed adjacency information in triangle strips, the specifics of this for the OpenGL API can be found in the specification for the `GL_EXT_geometry_shader4` extension [35].

4.4 Shadow Volume Extrusion

We now assume that all primitives coming into the geometry shader have been specified with correct adjacency information. Given this input to the geometry shader we can

make a significant alteration to the implementation of the shadow volume algorithm described in Section 2.3. The previous implementation used the CPU to determine each object's possible silhouette edges as seen from the light source. With access to triangle adjacency information on the GPU we can move the possible silhouette edge determination into a geometry shader. This means that geometry for the shadow quadrilaterals no longer needs to be sent from the CPU to the GPU each frame, reducing memory bandwidth usage and freeing up the CPU so that it can focus on other tasks.

Recalling that possible silhouette edges were defined in Section 2.3 as those edges shared by a back-facing and front-facing triangle with respect to the light source, this test can be implemented in the geometry shader as follows. For each incoming triangle that faces the light source, loop through each edge and see if the triangle adjacent to that edge is back-facing with respect to the light source. If it is, then emit an extruded semi-infinite hard shadow quadrilateral for that edge. It is important to realize that none of the triangles of the mesh are let through by this processing, only new extruded geometry is created, and many triangles in the original mesh will result in no shadow volume geometry at all.

The `main()` routine for this geometry shader is given in Listing 4.2, and the implementation of the `process_edge()` function is given in Listing 4.3. Note that when using triangles *with* adjacency information the value of `gl_VerticesIn` is 6.

Listing 4.2: Geometry shader for automatic possible silhouette edge detection and extrusion.

```
void main()
{
    // Determine the normal of this triangle
    vec3 n = cross(
        vec3(gl_PositionIn[2] - gl_PositionIn[0]),
        vec3(gl_PositionIn[4] - gl_PositionIn[0]));

    // Compute direction to the light
    vec3 l = light_position - vec3(gl_PositionIn[0]);

    // If this triangle is front facing then check
    // each edge's adjacent triangle to see if it's
    // back-facing, and if so, extrude a quad
    if (dot(n, l) > 0.0)
    {
        process_edge(vec3(gl_PositionIn[0]),
                     vec3(gl_PositionIn[2]),
                     vec3(gl_PositionIn[1]));
        process_edge(vec3(gl_PositionIn[2]),
                     vec3(gl_PositionIn[4]),
                     vec3(gl_PositionIn[3]));
        process_edge(vec3(gl_PositionIn[4]),
                     vec3(gl_PositionIn[0]),
                     vec3(gl_PositionIn[5]));
    }
}
```

Listing 4.3: Geometry shader that extrudes a semi-infinite shadow quadrilateral for an edge if its adjacent triangle is back facing with respect to the light source.

```
void process_edge(vec3 e0, vec3 e1, vec3 vAdjacent)
{
    // Determine normal of adjacent triangle
    vec3 n = cross(e1 - vAdjacent, e0 - vAdjacent);

    // Compute direction to the light
    vec3 l = light_position - vAdjacent;

    // Check if adjacent triangle is back facing
    if (dot(n, l) < 0.0)
    {
        // Create semi-infinite shadow quadrilateral
        extrude_edge(e0, e1);
    }
}
void extrude_edge(vec3 e0, vec3 e1)
{
    // Create semi-infinite shadow quadrilateral
    // Note that w = 0.0 for extruded vertices
    emit_vertex(vec4(e0, 1.0));
    emit_vertex(vec4(e0 - light_position, 0.0));
    emit_vertex(vec4(e1, 1.0));
    emit_vertex(vec4(e1 - light_position, 0.0));
    EndPrimitive();
}
void emit_vertex(vec4 v)
{
    // Transform vertex by the modelview projection
    // matrix before emitting it
    gl_Position = gl_ModelViewProjectionMatrix * v;
    EmitVertex();
}
```

Using this geometry shader means that all shadow volume extrusion is now done on the GPU, which considerably reduces the implementation burden of the algorithm. However, while this technique is useful, it does not necessarily translate into a real-world performance improvement. If the GPU is already overburdened then giving it this extra task will actually reduce performance. Also, we can not be certain that the implementation of geometry shading hardware on the GPU will perform possible silhouette edge determination faster than the CPU. In light of these observations, a full evaluation of this algorithm's performance is presented in Chapter 5.

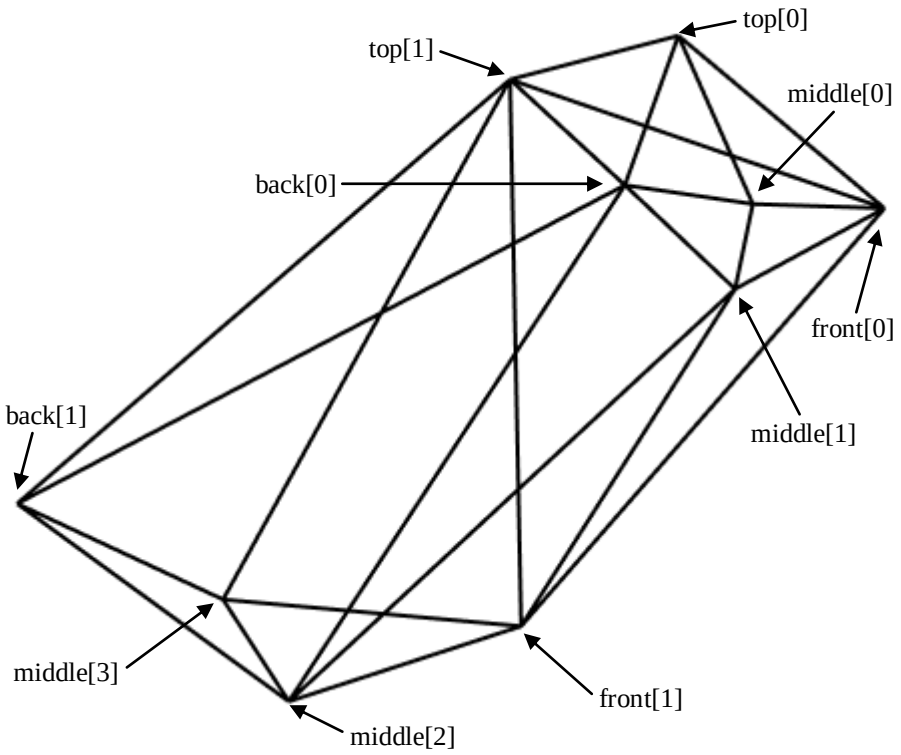

Figure 4.3: Penumbra wedge geometry.

4.5 Penumbra Wedge Extrusion

Having addressed the problem of extruding hard shadow quadrilaterals on the GPU using a geometry shader we now demonstrate that the same technique can be applied to achieve GPU construction of penumbra wedge geometry. The process of constructing penumbra wedge geometry was described in detail in Section 3.2.2, and we now give geometry shader that performs this part of the penumbra wedge algorithm.

The process of determining possible silhouette edges in the geometry shader is unchanged from the one described in Section 4.4, only the implementation of the `extrude_edge()` function changes. It is renamed to `extrude_penumbra_wedge()` to reflect the fact that it must now construct the extruded penumbra wedge geometry.

We will present the geometry shader code in stages, explaining the function of each stage along with its corresponding GLSL code. This assists with explanation and also breaks up the code as it would be overly long if presented as a single listing.

4.5.1 Vertex Positions

The first step in this process is to calculate the positions of the vertices that the penumbra wedge will be built out of. As illustrated in Figure 4.3, there are 10 vertices in a penumbra wedge: two at the top of the wedge, two vertices for each of the front and

back pieces, and four that make up the wedge's center plane. Recall that a wedge is made up of front, back, left and right planes, and these planes are now being represented by the following sets of co-planar wedge vertices:

1. Front plane: `top[0]`, `top[1]`, `front[0]` and `front[1]`.
2. Back plane: `top[0]`, `top[1]`, `back0]` and `back[1]`.
3. Left plane: `top[0]`, `back[0]`, `front[0]` and `middle[0]`.
4. Right plane: `top[1]`, `back[1]`, `front[1]` and `middle[3]`.

An additional center plane that contains all the `top` and `middle` vertices as well as the original hard shadow quadrilateral Q for the edge divides the wedge into front and back halves. The extra `middle[1]` and `middle[2]` vertices are the extruded vertices of Q and are added to the wedge geometry in order to avoid sub-pixel gaps appearing during rasterization. It should also be noted that triangles are added to the underside of the wedge halves so that the wedge is a completely enclosed volume, making it possible to use per-wedge stencil testing to mask out fragments outside the penumbra [36], which performs faster on some hardware compared to the fragment shader-based culling solution described in Section 3.2.4.

Listing 4.4 gives geometry shader code for the calculation of the `top[0]` and `top[1]` vertices using the method described in Section 3.2.2. Its two inputs, `e0` and `e1`, are the vertices of the possible silhouette edge.

Listing 4.4: Geometry shader for determining the top vertices of a penumbra wedge.

```
void extrude_penumbra_wedge(vec3 e0, vec3 e1)
{
    vec3 top[2], front[2], back[2], middle[4];

    // Compute rays from light to edge endpoints
    // as well as squared distances
    vec3 light0 = e0 - light_position;
    vec3 light1 = e1 - light_position;
    float distance0 = dot(light0, light0);
    float distance1 = dot(light1, light1);

    // Determine which endpoint is closest to the
    // light and set the top[] vertices
    if (distance0 < distance1) {
        light1 *= sqrt(distance0 / distance1);
        top[0] = e0;
        top[1] = light_position + light1;
    } else {
        light0 *= sqrt(distance1 / distance0);
        top[0] = light_position + light0;
        top[1] = e1;
    }
```

Assuming a spherical light source, calculation of the `front`, `back` and `middle` vertices is achieved by projecting the light sphere through the top of the penumbra wedge. The x and y extent vectors of the wedge's light plane (which contains the `top[0]` $\rightarrow$ `top[1]` edge) can be calculated directly by taking the x extent as the normalized vector from `top[0]` $\rightarrow$ `top[1]` and the y extent as the cross product of the x extent and a vector from any point on the edge to the light source. These extent vectors are then scaled by the radius of the light source and used as offsets that are applied to the light position. These offset light positions are then extruded through the wedge's `top` vertices to give the `front`, `back` and `middle` vertices. The geometry shader code for these procedures is given in Listings 4.5, 4.6 and 4.7.

Listing 4.5: Geometry shader for calculating the extents of a spherical light for a penumbra wedge.

```
// Compute the extents of the light plane that
// contains the top edge of the wedge
vec3 x_axis, y_axis;
x_axis = normalize(top[1] - top[0]);
y_axis = normalize(cross(x_axis, -light0));

// Adjust for light radius
x_axis *= light_radius;
y_axis *= light_radius;
```

Listing 4.6: Geometry shader for calculating an extruded vector from an offset light position through a point at the top of a penumbra wedge. Note that these vertices are extruded by a finite distance rather than undergoing an infinite extrusion.

```
vec3 extrude(vec3 top, vec3 light_offset)
{
    vec3 extent = light_position + light_offset;
    return top + normalize(top - extent) * range;
}
```

Listing 4.7: Geometry shader for calculating front, back and middle penumbra wedge vertices.

```
front[0]  = extrude(top[0],  x_axis + y_axis);
front[1]  = extrude(top[1], -x_axis + y_axis);
back[0]   = extrude(top[0],  x_axis - y_axis);
back[1]   = extrude(top[1], -x_axis - y_axis);
middle[0] = extrude(top[0],  x_axis);
middle[3] = extrude(top[1], -x_axis);
middle[1] = top[0] + normalize(light0) * range;
middle[2] = top[1] + normalize(light1) * range;
```

4.5.2 Primitive Output

Once the vertices of the wedge have been calculated they need to be correctly assembled into the final triangles that will be output from the geometry shader. With reference to Figure 4.3, specifying the vertices of these triangles is straightforward. The triangles for the front and back wedges, specified in a counter-clockwise winding order, are:

1. Front triangles = {`top[0]`, `top[1]`, `front[0]`}, {`front[0]`, `top[1]`, `front[1]`}, {`top[0]`, `front[0]`, `middle[0]`}, {`top[1]`, `middle[3]`, `front[1]`}, {`middle[3]`, `middle[2]`, `front[1]`}, {`front[1]`, `middle[2]`, `middle[1]`}, {`front[1]`, `middle[1]`, `front[0]`}, {`front[0]`, `middle[1]`, `middle[0]`}
2. Back triangles = {`top[0]`, `back[0]`, `top[1]`}, {`top[1]`, `back[0]`, `back[1]`}, {`top[0]`, `middle[0]`, `back[0]`}, {`top[1]`, `back[1]`, `middle[3]`}, {`middle[2]`, `middle[3]`, `back[1]`}, {`middle[2]`, `back[1]`, `back[0]`}, {`middle[2]`, `back[0]`, `middle[1]`}, {`middle[1]`, `back[0]`, `middle[0]`}

The winding order of these triangles is structured so that all triangles face outwards from the penumbra wedge. However, certain implementation variations reverse the winding order of the back wedge halves as part of rendering speed optimizations [37], in which case the triangle vertex orderings specified above would need to be altered. Also, if the implementation is rendering each half of the wedge separately then only one set of wedge triangles, either the front or back, will need to be generated for any given execution of the geometry shader.

4.5.3 Optimized Triangle Strips

Although emitting these triangles from the geometry shader would be correct output, the number of vertices emitted can be significantly reduced by outputting triangle strips from the geometry shader instead of separate triangles. With reference to Figure 4.3 the obvious candidates for this optimization are the pair of triangles that make up the front plane, the pair of triangles that make up the back plane, and the triangles that make up the wedge's bottom cap.

The naïve implementation outputs 16 triangles for each extruded wedge for a total of 48 separate vertices. Refactoring the output code to generate triangle strips where possible reduces the number of output vertices to 32, a saving of 33%. It may be possible to further reduce this number through a more complex triangle strip configuration for the penumbra wedge. An off-line triangle strip creation tool such as NVIDIA's NVTriStrip [38] could potentially be used to compute an optimal configuration.

4.6 Summary

In this chapter we have shown how the new programmable geometry shaders and adjacency primitive features present in the latest generation of graphics hardware can be used for GPU acceleration of possible silhouette edge determination and subsequent extrusion of shadow volume and penumbra wedge geometry. The ability to do this type of computation on the GPU is a very recent development, we know of no existing implementation that is similar to ours. In Chapter 5 we present results of performance testing for our implementation of this novel approach.

Chapter 5

Results

In this chapter we present two sets of results based on our implementation of the two soft shadowing algorithms covered in Chapter 3 and of the geometry shader methods from Chapter 4.

The first set of results consists of detailed performance measurements for the percentage closer soft shadows algorithm and the penumbra wedge soft shadows algorithm. We focus on how the performance of the two approaches compares in a standardized test environment. The important factors influencing the performance of each of these algorithms have been covered in Sections 3.1 and 3.2, along with the tradeoffs they offer with regard to sacrificing quality for increased performance, and vice versa.

The second set of results focusses on the performance characteristics of using geometry shaders for both hard and soft shadow rendering as described in Chapter 4. These results are of particular interest as they offer an early insight into how this new technology performs in the first generation of consumer graphics hardware that implements it.

5.1 Test Environment

The hardware and software specifications of the test environment used for performance testing of the algorithms were:

- Processor: AMD Athlon 64 3500+
- System memory: 1GB DDR400
- Graphics processor: NVIDIA GeForce 8600 GTS (PCI-Express)
- Video memory: 256MB
- Operating system: Microsoft Windows XP Professional SP2
- Graphics driver: NVIDIA Forceware 163.71 (WHQL Approved)

In terms of overall system performance this would be considered a mid-range hardware setup, although the graphics processor is more up-to-date and supports all of the latest generation hardware features necessary for this testing (e.g. geometry shaders).

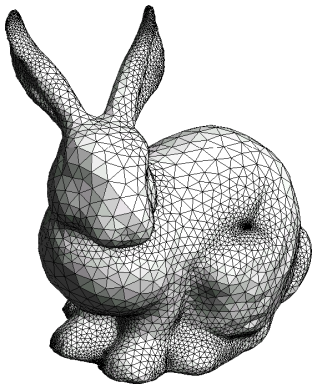

Figure 5.1: The Stanford bunny model used for the performance analyses.

The implementations were also tested on the Ubuntu Linux 7.10 operating system and worked successfully with both the GCC 4.2 and the Intel C++ 10.0 compilers. However, the overall performance of GPU-intensive applications on Linux-based systems is typically lower than on Microsoft Windows XP, and we defer a detailed comparative analysis of relative platform performance for the soft shadowing algorithms to a later date.

5.2 Scene Setup

When conducting performance analysis it is important to use a standardized testing setup. In light of this we used the Stanford bunny model for all performance tests. This model, pictured in Figure 5.1, has been used in the analysis of a wide variety of computer graphics algorithms since its introduction in 1994 [39]. It consists of 35497 vertices and 69451 triangles. The Stanford bunny model was scaled so that its axis-aligned bounding box was exactly 5.0 units in height.

Additional geometry was added in the form of a floor plane on which the bunny model sits. For performance testing all graphical processing not essential for shadow determination such as per-pixel lighting, surface texturing, and full screen anti-aliasing was disabled. This was done to get as close as possible to measuring the real-time performance of the shadowing algorithm in isolation.

5.3 Algorithm Performance

Before giving performance results we first describe the exact setup used for each algorithm.

For the percentage closer soft shadow algorithm a shadow map resolution of 512×512 pixels was used as it gave sufficiently high-quality results in our test scene. The shadow map was rendered from the light's position pointing directly at the bunny model and the viewing frustum planes were pulled in as far as possible without causing clipping of the model. This optimizes the use of the available shadow map resolution. The

Resolution	Percentage Closer	Penumbra Wedge
640×480	78	54
800×600	66	46
1024×768	49	36
1280×1024	35	29

Table 5.1: Results of performance testing on the percentage closer soft shadow algorithm and the penumbra wedge soft shadow algorithm at different output resolutions. The units are the number of frames rendered per second.

Model resolution	Extrusion Location	Shadow Volume	Penumbra Wedge
Full	CPU	126	77
Full	GPU	94	62
Reduced	CPU	242	132
Reduced	GPU	280	159

Table 5.2: Results of performance testing on the geometry shader shadow volume and penumbra wedge extrusion algorithms at two different model resolutions. The units are the number of frames rendered per second.

blocker search sample grid size was fixed at 6×6 and the percentage closer filter sample grid size was also 6×6. The shadow map was fully recomputed every frame even though the scene was static.

For the penumbra wedge soft shadow algorithm CPU extrusion of the penumbra wedge geometry was used, and penumbra fragment clipping was performed on a per-fragment basis as described in Section 3.2.4. Wedge geometry was fully recomputed every frame even though the scene was static. Inner and outer wedge halves were rendered separately to the visibility buffer.

For both algorithms a single spherical light source of radius 1.0 was used. It was positioned 5.0 units directly above the center of the bunny model's bounding box. The camera was positioned at the height of the light source, offset 10.0 units along the z axis, and pointed at the center of the bunny model's bounding box. Performance readings were taken with the scene sitting statically in this state. We waited for the frame rate to settle down following the loading of the scene and then counted the total number of frames rendered in the following 30 seconds and divided by 30 to get the final performance number in frames per second.

The measured performance numbers for both algorithms at four different output resolutions are given in Table 5.1. As expected, increasing output resolutions reduces the frame rate for both algorithms. The percentage closer soft shadow algorithm outperforms the penumbra wedge algorithm between by between 20 and 40% at all resolutions under our testing conditions. These performance results are evaluated and discussed in Section 5.5.

5.4 Geometry Shader Performance

We now present the results of the performance analysis done on the geometry shader shadow geometry construction techniques that were described in Chapter 4. Because the goal was to evaluate the performance characteristics of this technique in isolation we reduced the output resolution to 1×1 in order to eliminate bottlenecks associated with rasterization and fragment processing. Furthermore, in order to get an idea of how the geometry shader implementation scales with different mesh sizes we tested performance with both the original Stanford bunny model and a reduced quality version of the model that has only 4997 triangles. The mesh reduction was done with QSlim version 2.1 [40].

The results of the performance comparison are given in Table 5.2, and contain several surprising measurements. In the case of standard shadow volumes on the full resolution mesh the geometry shader reduced performance by 25%, but for the reduced resolution mesh it increased the performance by 16%.

When evaluating these performance results for the penumbra wedge algorithm it should be pointed out that they are going to be inflated due to the 1×1 output resolution. The penumbra wedge algorithm leans heavily on complex fragment processing and so the performance numbers in Table 5.2 do not reflect a standard real-time usage scenario. This was done deliberately to gain insight into the performance of the geometry shader hardware and specifically the part it plays in shadowing performance when used for automatic extrusion. Looking at the performance results for the penumbra wedge algorithm we see that it has similar trends to the results for the standard shadow volume algorithm. The geometry shader decreases performance for the high resolution mesh by 19% and increases it for the reduced resolution mesh by 21%.

We now discuss and evaluate the possible reasons for the observed performance results.

5.5 Discussion

Although both the percentage closer soft shadow algorithm and the penumbra wedge soft shadow algorithm share the goal of soft shadow generation, the ways in which they achieve this diverge markedly. This makes direct comparison challenging and difficult to draw strong conclusions from. Many important factors that influence what algorithm is most suitable in a given situation can't be represented in a set of performance numbers, and indeed performance is usually only one of many criteria to consider. For example, shadow map resolution in the percentage closer soft shadow algorithm has no direct equivalent in geometry-based shadowing algorithms, so there is no well-defined way to define what shadow map resolution to use for an algorithm comparison other than what produces visually similar results in the test scenario. The 512×512 resolution shadow map used in measuring the performance results in Table 5.1 is probably not sufficient for more complex scenes, and increasing it would likely change the performance picture considerably. Similarly, if an application needs to make heavy use of omnidirectional light sources then the penumbra wedge algorithm is almost certainly going to be the clear performance winner.

In spite of these types of problems, the results in Table 5.1 do paint a clear picture of how the two algorithms scale with output resolution, as well as giving an indication

of the level of runtime performance that can be expected from them.

Analyzing the second set of results in Table 5.2 poses some interesting questions regarding geometry shader performance. An explanation for the divergent behavior of the low and high resolution meshes is needed. What could cause high resolution meshes to run more slowly with geometry shader extrusion and also cause low resolution meshes to run faster with the same geometry extrusion program enabled? To answer this question we need to look more closely at the interactions between the CPU and GPU subsystems. In order to get a performance improvement from geometry shaders the time taken for the execution of the geometry shader needs to be less than the time that would have been taken for the CPU to do silhouette determination and upload the result to the GPU.

Consider that in our testing the CPU had relatively little work to do, its only real task was to direct the GPU. With geometry shaders enabled the CPU is spending the majority of its time waiting for the GPU to complete a task, and therefore using this spare processing time to determine object silhouettes is likely to be a good division of labor between the two processing units that improves performance. However, for less complex objects the overhead involved in the transfer of data from the CPU to the GPU will be higher than the time taken for the GPU to compute the data directly itself, and so in this case the geometry shader is the better option. This would not necessarily be the case if the CPU was overloaded with other work to do such as calculating physics, A.I., or other computations. In this scenario geometry shaders may be the sensible choice that reduces stress on the CPU, however more testing is needed to determine the scenarios in which this is the case.

Finally, it should also be noted that geometry shaders are a first generation hardware feature, which likely means the hardware implementation is not yet completely optimal and that future graphics processors will possibly bring significant performance improvements. There are also other graphics processors available today that have significantly higher performance than the one used in our testing, specifically the NVIDIA GeForce 8800 Ultra. It is likely that this graphics processor would have markedly different geometry shader performance characteristics compared to our test hardware, which could force us to rethink our analysis.

Chapter 6

Further Work

In this chapter we briefly identify further work relating to real-time soft shadowing algorithms that could be pursued either by ourselves or by other interested researchers.

6.1 Geometry Shaders

Our treatment of using of geometry shaders for the calculation of real-time soft shadows is only a first step into this new area. More investigation into the performance characteristics and best-practices of the geometry processor is needed, as well as the discovery of new ways to utilize the extra control that geometry shaders allow over the graphics pipeline.

6.2 Shadow Cube-mapping

One of the drawbacks of shadow map based algorithms is that omnidirectional light sources are often tricky to implement. However, the latest generation of hardware features the ability to render an entire cube-map in one pass, meaning that a complete shadow cube-map can now be rendered much more easily than on previous hardware. This is directly relevant to soft shadowing algorithms that are based on shadow mapping, which includes percentage closer soft shadows, as it makes the problem of omnidirectional light sources a thing of the past.

6.3 Silhouette Level of Detail

As discussed in Section 1.3, the inherent fuzziness of soft shadows means that the geometric complexity of the shadow casting object can often be reduced without adversely affecting shadow quality. This leads to the possibility of developing level-of-detail algorithms that are specifically designed to reduce mesh complexity in a way that minimally impacts resulting soft shadow quality, rather than optimizing mesh reduction for direct rendering. Designing algorithms specifically for this use pattern could result in better soft shadowing quality and higher performance of soft shadowing algorithms.

Chapter 7

Conclusions

In summary, we implemented two real-time soft shadowing algorithms, the percentage closer soft shadow algorithm and the penumbra wedge soft shadow algorithm, and analyzed their relative performance characteristics in a standardized testing setup. Details of both algorithms as well as the earlier shadowing algorithms on which they are built were given and implementation considerations were discussed.

Our performance testing found the percentage closer soft shadows algorithm to be the faster of the two algorithms across a variety of different output resolutions. However, we noted that simply because both algorithms were achieving comparable soft shadow rendering in our standardized test scene that did not necessarily make their performance and scalability directly comparable. Image-based techniques like the percentage closer soft shadow algorithm are a fundamentally different way of thinking about soft shadow generation than geometry-based approaches such as the penumbra wedge soft shadow algorithm, and in real-world usage scenarios factors such as the nature of the light sources being used can mean one algorithm will be preferred over another regardless of a 20% performance delta.

It is certainly possible that different scene, camera or lighting setups could significantly change our performance results, and we can't be certain that the observed trends hold for other graphics processors without explicit testing.

Our implementation of the penumbra wedge soft shadow algorithm had the novel ability to construct penumbra wedge geometry directly on the GPU using geometry shaders, and we gave a detailed performance analysis and evaluation of this new programmable stage in the graphics pipeline. There is still much work to be done in order to fully explore the possibilities that geometry shading allows. We analyzed the performance of our shadow algorithm implementations that were altered to use geometry shaders, and noted a number of interesting performance characteristics specific to geometry shaders. A possible explanation for the results we observed was put forward, and further in-depth testing would be required to verify whether our theory is correct.

Despite some problems, our analysis of soft shadowing algorithms and the utilization of geometry shading hardware to accelerate shadow rendering has yielded useful new results that spark a number of new areas of research interest. The application of geometry shaders to a wide variety of rendering tasks is likely to become significantly more prevalent in the near future, and real-time soft shadowing algorithms will continue to be an important area of research that pushes this boundary.

Bibliography

[1] J.-M. Hasenfratz, M. Lapierre, N. Holzschuch, and F. Sillion, "A survey of real-time soft shadows algorithms," *Computer Graphics Forum*, vol. 22, no. 4, pp. 753–774, 2003.

[2] U. Assarsson and T. Akenine-Möller, "A geometry-based soft shadow volume algorithm using graphics hardware," *ACM SIGGRAPH 2003 Papers*, pp. 511–520, 2003.

[3] M. Schwarz and M. Stamminger, "Bitmask soft shadows," *Computer Graphics Forum*, vol. 26, no. 3, 2007.

[4] C. Wyman and C. Hansen, "Penumbra maps: Approximate soft shadows in real-time," in *Proceedings of the 14th Eurographics Workshop on Rendering*, 2003, pp. 202–207.

[5] E. Eisemann and X. Décoret, "Plausible image based soft shadows using occlusion textures," in *Proceedings of the Brazilian Symposium on Computer Graphics and Image Processing*, 2006, pp. 155–162.

[6] V. Forest, L. Barthe, and M. Paulin, "Realistic soft shadows by penumbra-wedges blending," in *Proceedings of the 21st ACM SIGGRAPH/Eurographics Symposium on Graphics Hardware*, 2006, pp. 39–46.

[7] E. Chan and F. Durand, "Rendering fake soft shadows with smoothies," in *Proceedings of the 14th Eurographics Workshop on Rendering*, 2003, pp. 208–218.

[8] J.-F. St-Amour, E. Paquette, and P. Poulin, "Soft shadows from extended light sources with penumbra deep shadow maps," in *Proceedings of Graphics Interface 2005*, 2005, pp. 105–112.

[9] M. Sattler, R. Sarlette, T. Mücken, and R. Klein, "Exploitation of human shadow perception for fast shadow rendering," in *Proceedings of the 2nd Symposium on Applied Perception in Graphics and Visualization*, 2005, pp. 131–134.

[10] L. Wanger, "The effect of shadow quality on the perception of spatial relationships in computer generated imagery," in *Proceedings of the 1992 Symposium on Interactive 3D Graphics*, 1992, pp. 39–42.

[11] D. Kersten, P. Mamassian, and D. Knill, "Moving cast shadows and the perception of relative depth," Max Planck Institute for Biological Cybernetics, Tech. Rep. 6, 1994.

[12] P. Mamassian, D. Knill, and D. Kersten, "The perception of cast shadows," *Trends in Cognitive Sciences*, vol. 2, no. 8, pp. 288–295, 1998.

[13] A. Woo, P. Poulin, and A. Fournier, "A survey of shadow algorithms," *IEEE Computer Graphics and Applications*, vol. 10, no. 6, pp. 13–32, 1990.

[14] L. Neumann and A. Neumann, "Radiosity and hybrid methods," *ACM Transactions on Graphics*, vol. 14, no. 3, pp. 233–265, 1995.

[15] T. T. Yu, J. Lowther, and C. K. Shene, "Photon mapping made easy," in *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, 2005, pp. 201–205.

[16] J. F. Blinn, "Me and my (fake) shadow," *IEEE Computer Graphics and Applications*, vol. 8, no. 1, pp. 82–86, 1988.

[17] E. Haines, "Soft planar shadows using plateaus," *Journal of Graphics Tools*, vol. 6, no. 1, pp. 19–27, 2001.

[18] S. Brabec, T. Annen, and H. P. Seidel, "Shadow mapping for hemispherical and omnidirectional light sources," in *Proceedings of Computer Graphics International*, 2002, pp. 397–408.

[19] B. Osman, M. Bukowski, and C. McEvoy, "Practical implementation of dual paraboloid shadow maps," in *Proceedings of the 2006 ACM SIGGRAPH Symposium on Videogames*, 2006, pp. 103–106.

[20] W. T. Reeves, D. H. Salesin, and R. L. Cook, "Rendering antialiased shadows with depth maps," *ACM SIGGRAPH Computer Graphics*, vol. 21, no. 4, pp. 283–291, 1987.

[21] M. Stamminger and G. Drettakis, "Perspective shadow maps," in *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, 2002, pp. 557–562.

[22] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Priniples and Practice*, 2nd ed. Addison-Wesley, 1995.

[23] F. C. Crow, "Shadow algorithms for computer graphics," in *Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques*, 1977, pp. 242–248.

[24] T. Heidmann, "Real shadows, real time," *Iris Universe*, vol. 18, pp. 23–31, 1991.

[25] C. Everitt and M. J. Kilgard, "Practical and robust stenciled shadow volumes for hardware-accelerated rendering," NVIDIA Corporation, Tech. Rep., 2002.

[26] M. Kilgard, "Improving shadows and reflections via the stencil buffer," NVIDIA, Tech. Rep., 1999.

[27] C. Everitt and M. J. Kilgard, "Optimized stencil shadow volumes," NVIDIA, Tech. Rep., 2003.

[28] M. McCool, "Shadow volume reconstruction from depth maps," *ACM Transactions on Graphics*, vol. 19, no. 1, pp. 1–26, 2000.

[29] R. Fernando, "Percentage-closer soft shadows," *ACM SIGGRAPH 2005 Sketches and Applications*, 2005.

[30] T. Akenine-Möller and U. Assarsson, "Approximate soft shadows on arbitrary surfaces using penumbra wedges," in *Proceedings of the 13th Eurographics Workshop on Rendering*, pp. 297–306.

[31] U. Assarsson and T. Akenine-Möller, "Occlusion culling and z-fail for soft shadow volume algorithms," *The Visual Computer: International Journal of Computer Graphics*, vol. 20, no. 8–9, pp. 601–612, 2004.

[32] R. Rost, *OpenGL Shading Language*, 2nd ed. Addison-Wesley, 2006.

[33] NVIDIA, "Nvidia geforce 6 series specifications," Tech. Rep., 2004.

[34] F. Luna, *Introduction to 3D Game Programming with Direct X 9.0c: A Shader Approach*. Wordware Publishing, 2006.

[35] "GL_EXT_geometry_shader4," NVIDIA, Tech. Rep., 2007.

[36] U. Borgenstam and J. Svensson, "A soft shadow rendering framework in OpenGL," Master's thesis, Department of Computer Engineering, Chalmers University of Technology, 2004.

[37] E. Lengyel, "Advanced stencil shadow and penumbral wedge rendering," Game Developers Conference Presentation, Tech. Rep., 2005.

[38] NVIDIA, "NVTriStrip Library," 2004. [Online]. Available: http://developer.nvidia.com/object/nvtristrip_library.html

[39] G. Turk and M. Levoy, "Zippered polygon meshes from range images," in *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techinques*, 1994, pp. 311–318.

[40] M. Garland, "QSlim mesh simplification tool v2.1," http://graphics.cs.uiuc.edu/~garland/software/qslim.html, 2004.